Université Paris 7 – Denis-Diderot
UFR d'Informatique

Thèse

pour l'obtention du diplôme de
Docteur de l'Université Paris 7, spécialité informatique

Réplications Distribuées pour la Définition des Interactions de Jeux Massivement Multi-Joueurs

Anne-Gwenn Bosser

Présentée et soutenue publiquement le
18 Novembre 2005

Directeur de thèse
Emmanuel Chailloux

Jury

M. Frédéric Boussinot	Rapporteur
M. Emmanuel Chailloux	Directeur de thèse
M. Guy Cousineau	Président
M. Ryohei Nakatsu	Examinateur
M. Stéphane Natkin	Rapporteur

C'est une maison tellement grande l'absence
Pablo Neruda

Remerciements

Tout d'abord, merci à Emmanuel Chailloux qui suit depuis de longues années maintenant mes tribulations estudiantines et professionnelles, et grâce à qui j'ai commencé cette thèse (bien sûr, j'ai parfois dit «à cause de qui», c'est long et douloureux de temps en temps une thèse).

Je tiens également à remercier ceux qui m'ont fait l'honneur d'accepter de participer au jury de soutenance.

- Frédéric Boussinot, dont les travaux ont inspiré les miens, et Stéphane Natkin, avec qui je partage mon intérêt pour les jeux-vidéo en ligne, ont accepté de rapporter ce travail. J'avoue que j'en suis très fière.
- Guy Cousineau m'a soutenue pour réaliser cette thèse dans de bonnes conditions, tandis que Ryohei Nakatsu m'a accueillie dans son laboratoire d'Osaka à un moment ou je commençais à manquer d'inspiration, et je l'y ai retrouvée.

Le laboratoire PPS a été un lieu de travail chaleureux et accueillant au cours de ces trois années et des grosses poussières. Un merci tout particulier à son directeur Pierre-Louis Curien, qui malgré ma thématique de recherche originale pour le laboratoire, m'a fait me sentir à l'aise au sein de ce cocon privilégié, et à Odile Ainardi (aka «La magicienne des problèmes administratifs»). Je ne vais pas faire la liste de tous les membres du laboratoire qui feront les bons souvenirs parce que ça fait du monde voire des pages. Un merci tout spécial quand même à la bande de thésards de PPS passée et actuelle, et à celle du couloir d'en face. J'ai juste eu à évoquer le sujet pour qu'autour de la table à café, en quelques secondes, mon manuscrit soit divisé et distribué pour la guerre aux coquilles. Je crois que c'est quand même Caroline Priou et Claire David qui ont reçu les parties les plus difficiles à lire. A charge de revanche c'est promis.

Au chapitre relectures, ainsi que *hotline* latex, c'est quand même Sylvain Baro qui gagne haut-la-main ma reconnaissance éternelle. La pertinence de

ses remarques, autant sur le fond que la forme de ce manuscrit m'a permis de l'améliorer d'une manière sensible. Mes parents ont également effectué dans l'urgence une relecture qui a permis de juguler mon orthographe et ma grammaire quantique (parfois elles sont là, parfois non), tout en agrémentant les marges de leur copie de recettes de cuisine et jeux de mots tordus. Pour les articles en anglais qui ont rythmé ces trois années, mes écarts de langages ont été sévèrement réprimés par Francisco Alberti, Isabelle Chelley et Russ Harmer. Je suis sûre que quelques comités de lecture de par le monde vous sont également très reconnaissants.

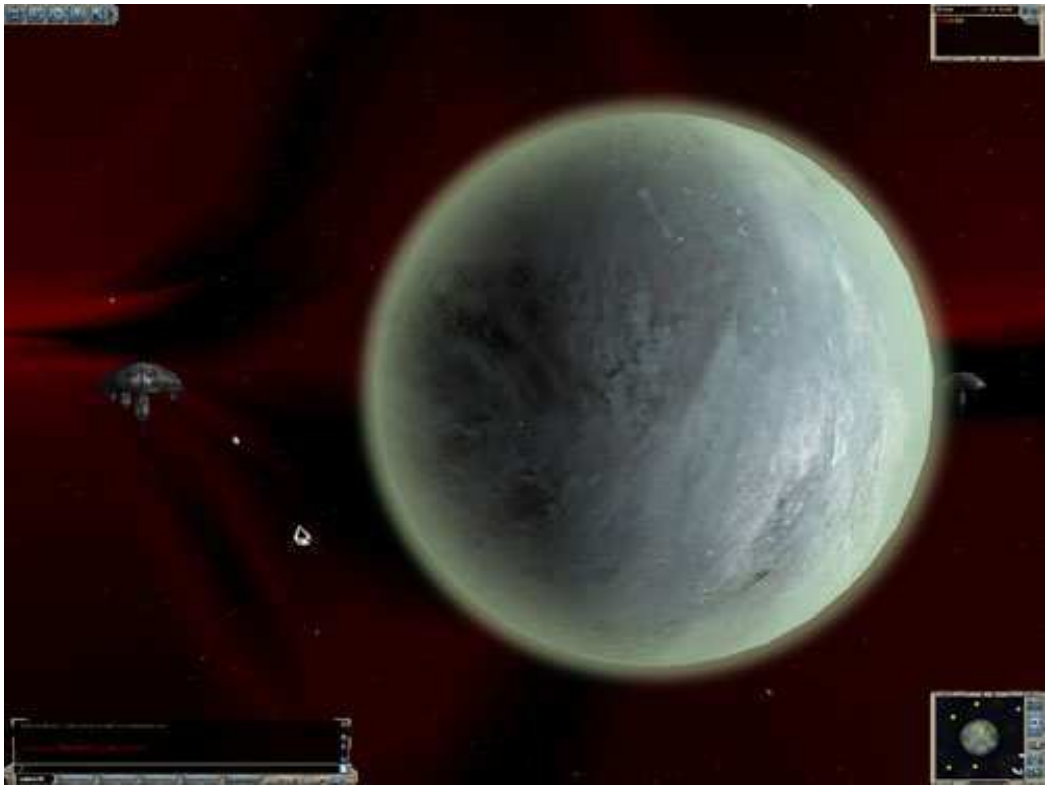
Merci également aux étudiants que j'ai rencontrés lors des enseignements à Paris 7. Lorsqu'on cherche, on doute, et vous enseigner était parfois une vraie bulle d'air et toujours un moment agréable dans mon quotidien. Merci à Juliusz Chroboczek, nos discussions très animées vont me manquer. Non je ne blague pas. Et aussi à Vincent Padovani, qui est le genre d'enseignant que j'aimerais devenir.

Vincent Padovani et Philippe Codognet ne s'en sont sans doute pas rendus compte, mais ils ont tous les deux prononcé quelques mots, exactement ceux qu'il fallait, à des moments où j'avais vraiment besoin de les entendre. Je ne vais pas les rapporter ici, et les laisser se torturer l'esprit.

Merci à mes amis et à ma famille, et à ma tribu du far-west, pour le soutien moral indéfectible. Merci à Marc, pour toute la paperasse de 2002, à Ariane pour nos bobuns en terrasse, à Cécile, pour avoir tenté avec courage et abnégation de me faire adopter une vie saine malgré le travail, à Francisco pour sa disponibilité, et à Juliette et Sylvain pour les caipis. Merci aussi à Nadine pour les tuyaux sur l'expatriation, à Benjamin, François et Raphaël pour les blagues douteuses de la pause café, et à Odile pour son sourire. Merci à mes tribus virtuelles, surtout aux *Brumes Lysergiques*. Maintenant, j'aurai peut-être plus de temps pour jouer.

Les copies d'écran de jeux utilisées dans le premier chapitre de ce manuscrit sont sauf mention contraire les oeuvres de Mimoza Honeyskul, Sam, Chorée et Ranarama (et non, tous ne sont pas moi).

Enfin et surtout, merci Mathieu, pour avoir supporté de manière constructive et sans broncher états d'âmes et nuits blanches. Promis, ça peut pas être pire.



Résumé

Les Jeux Massivement Multi-Joueurs sont des applications distribuées sur Internet dans lequels on retrouve des problématiques de persistance, de sécurité, de temps-réel, de passage à l'échelle, et d'utilisation critique des ressources des machines et du réseau. Nous proposons un cadre pour la réalisation de telles applications afin de favoriser la mise au point de *game-play* innovants en permettant une mise au point très fine des interactions. Nous décrivons les techniques actuelles et démontrons comment chaque solution pour la réalisation d'une interaction donnée est fortement liée à la description fonctionnelle de cette interaction dans le cadre du game-play considéré. Notre proposition consiste en un outil de prototypage basé sur un framework doté d'une sémantique simple pour simplifier le développement, mais permettant de gérer très finement les ressources bas-niveau afin de ne pas manquer de généricité. L'outil est destiné à être utilisé dans un cadre réaliste de méthodologie de développement basée sur le raffinement successif de prototypes permettant de valider au plus tôt les choix techniques. Nous présentons le framework que nous avons développé, qui définit un modèle très fin de réplication des données représentant le monde virtuel le long de l'application distribuée. La sémantique utilisée repose sur un modèle d'exécution coopératif et reproductible, dont nous donnons la formalisation des principaux traits sous une forme opérationnelle. Nous décrivons l'organisation du code produit, ainsi que la manière dont le framework s'inscrit dans notre proposition finale et détaillons un exemple complet pour illustrer son utilisation.

Abstract

Massively Multiplayer Online Games are applications distributed over the web for which the issues of persistence, security, real-time response, scalability and the critical use of both processor and network resources, are

problematic. We propose a framework for the construction of such applications, that favors the fine-tuning of innovating game-plays by allowing a very fine-grained adjustment of the interactions involved. We describe the techniques currently in use and show how each of the existing solutions for the construction of a given interaction is closely related to the functional description of that interaction, considered in the context of a specific game-play. Our proposal comprises a prototyping tool based on a framework having a simple semantics in order to facilitate programming. The detailed management of low-level resources is however possible in order to keep its generality. This tool has been designed to be applied as part of a concrete programming methodology based on the stepwise refinement of prototypes, thus allowing the validation of technical decisions as early as possible. We present the framework we have developed, for which we defined a very detailed replication model of data representing the virtual world in the distributed application. The semantics adopted is based on an execution model that is both cooperative and reproducible, for which we provide the formalization of its most important features in operational form. We describe the structure of the code and show how the framework can be used in the final solution. As a matter of illustration, we conclude with the complete study of a simple example.

Table des matières

Table des figures

Introduction

Une autre vie sociale

Depuis que les ordinateurs existent, ils ont été détournés de leur vocation première par les premiers développeurs de jeux-vidéo : dès 1972, Pong, commercialisé par Atari remporte un franc succès en permettant aux possesseurs d'ordinateurs personnels de jouer au ping pong avec des bâtons en guise de raquettes. Ensuite, les ordinateurs ont commencé à se connecter en réseaux. Tout naturellement, les joueurs se sont mis à jouer ensemble au travers de ces réseaux. Cela n'a donc pas été une surprise lorsqu'Internet a commencé à être utilisé à des fins vidéo-ludiques...

Avec la démocratisation des connexions à haut-débit, et la généralisation des accès à Internet chez les particuliers, un type nouveau de jeu-vidéo est né, le jeu massivement multi-joueurs (voir glossaire). Le développement de ces nouvelles distractions ludiques est exponentiel ces dernières années. La société Wanadoo, très satisfaite de son jeu massivement multi-joueurs gratuit *La quatrième prophétie*, a présenté dans son rapport d'activité 2001 [102] ce nouveau loisir comme une *Killer-Application* (voir glossaire) pour le développement des connexions à haut-débit. Suite à ce succès, la société a décidé en 2002 d'exploiter un nouveau jeu massivement multi-joueurs, *Dark Ages of Camelot* [41] en Europe, faisant désormais payer un abonnement aux joueurs. C'est à ce moment là que de loisir confidentiel, réservé à une élite de *Hardcore-gamers* (voir glossaire), les jeux massivement multi-joueurs sont passés au statut loisir grand-public en France.

Nous avons même assisté pendant la durée qui s'est écoulée depuis le début de cette thèse, à l'été 2002, à une explosion considérable du nombre

d'adeptes. Au fil de ces trois années, l'auteur a rencontré de moins en moins de difficultés à expliquer son sujet d'étude à ses interlocuteurs de moins en moins dubitatifs.

Plus encore que les jeux-vidéo traditionnels, les jeux massivement multi-joueurs sont souvent présentés comme des loisirs addictifs, donnant lieu à de nombreuses controverses sur leurs conséquences sur la vie sociale de leurs adeptes. Les médias nous mettent régulièrement en garde contre leurs dangers (précisant parfois l'existence d'une consultation spéciale pour désintoxiquer les drogués des cyber-mondes), et on étudie avec intérêt ou amusement leurs impacts sociologiques [58, 61] et l'impact potentiel des économies virtuelles sur l'économie du monde réel [22]. En effet, l'expression *Monde Virtuel* n'a jamais été aussi bien utilisée que pour désigner un jeu massivement multi-joueurs. Ces jeux sont des univers persistants, ils continuent à évoluer, le temps s'y écoule, des événement s'y produisent que le joueur y soit connecté ou non. D'autres joueurs continuent à y évoluer, pendant ce temps. Bref, le monde virtuel continue d'exister même si le joueur n'y est pas, il n'y a pas de fin à la partie.

Dans un jeu massivement multi-joueurs, chaque joueur incarne un *avatar* (voir glossaire). Ce dernier, qui est généralement un personnage, est le même d'une connexion à l'autre. Au fil du temps passé à jouer, l'avatar progresse dans le monde en terme de possessions virtuelles et de savoir-faire durement gagnés. L'avatar acquiert une personnalité, et il est vrai que l'aspect social joue là un grand rôle. On ne joue pas seul : des liens se créent entre avatars, ou entre joueurs on ne sait plus trop bien, certains avatars acquièrent un certain prestige au sein de la communauté. Au final, un joueur de jeu massivement multi-joueurs s'attache à son avatar et à sa vie virtuelle.

Un jeu massivement multi-joueurs n'est pas qu'un simple jeu-vidéo, c'est une autre vie sociale.

Des applications complexes

Les jeux massivement multi-joueurs sont un sujet fascinant pour un chercheur en informatique. En effet, la réalisation d'une telle application regroupe à elle seule à peu près toutes les problématiques des applications distribuées

sur Internet.

Ce sont des applications qui doivent réagir en temps réel aux actions des utilisateurs. Or, même si la transmission d'une information par Internet semble rapide, surtout lorsqu'on a accès à une connexion à haut-débit, elle est loin d'être instantanée. L'exemple tarte à la crème du développeur de jeu en ligne est celui de l'aller-retour d'un paquet d'informations entre, mettons, Paris et Melbourne : un calcul simple montre que même dans un réseau théorique parfait, où les informations circuleraient à la vitesse de la lumière et ne seraient retardées par aucun routeur physique les aiguillant sur le réseau, le trajet prendrait plus de 100ms. Or, 100ms, c'est le temps moyen estimé du réflexe humain. Il y a donc une limite purement physique avec laquelle il faut composer lorsqu'on réalise un jeu massivement multi-joueurs. De plus, Internet est un réseau hétérogène qui ne fournit aucune garantie quant à la vitesse d'acheminement des données. Il est sujet à des problèmes de congestions, lorsque le réseau est saturé par les informations qui y circulent par endroit. L'application devra donc essayer de masquer au mieux les temps de transmission excessifs aux joueurs, afin qu'ils gardent l'impression d'interagir naturellement avec le jeu et les autres joueurs, et qu'ils vivent une immersion totale dans le monde virtuel.

Les jeux massivement multi-joueurs sont également comme nous l'avons vu précédemment des applications persistantes, de véritables mondes virtuels qui continuent à exister et à évoluer, que le joueur y soit présent ou non. Le degré de sécurité nécessité par l'application est donc important, car le pouvoir de nuisance d'un tricheur modifiant l'état du monde à son avantage, ou d'un pirate informatique détruisant ou introduisant des données corrompues, peut avoir des conséquences à long terme sur l'économie ou la mécanique des mondes virtuels. De plus, les joueurs sont également des clients qui payent chaque mois, et comme nous l'avons vu, s'attachent très facilement à leur avatar et possessions virtuelles durement et justement gagnées. Un problème de sécurité ou de triche n'a donc pas que des conséquences sur le monde virtuel, mais risque aussi de faire chuter rapidement les bénéfices du jeu.

Enfin, les jeux massivement multi-joueurs doivent pouvoir gérer un nombre très important de joueurs simultanés, et donc être robustes vis-à-vis d'un passage à l'échelle en terme de nombre de joueurs. Bien sûr, cela signifie que le jeu doit permettre à un grand nombre de joueurs d'être connectés simultanément. Mais cela signifie aussi que si une grande activité règne dans une

région donnée du monde virtuel, où un grand nombre de joueurs ont décidé de rassembler leurs avatars, chaque joueur doit pouvoir continuer à évoluer de manière satisfaisante, et pour cela recevoir en grande quantité les informations sur ce qui se déroule autour de lui avec assez de rapidité pour continuer à profiter agréablement de l'expérience.

La réalisation d'un jeu massivement multi-joueurs est donc un projet complexe et sensible, difficile à mettre au point d'un point de vue technique. La meilleure preuve en est le nombre de jeux massivement multi-joueurs dont le développement a été commencé, mais jamais terminé. Certains projets ont été abandonnés après des années d'investissements de la part des sociétés qui s'y sont essayées. D'autres sociétés ont fait faillite avant que ces projets coûteux arrivent à leur terme. Le sujet de cette thèse a d'ailleurs été inspiré par les difficultés rencontrées dans la réalisation d'une telle application par un précédent employeur... Nous consacrerons un chapitre entier de ce manuscrit à la description de ces difficultés et aux moyens de les résoudre.

De plus, il y a encore peu, un jeu se développait à trois (ou deux, pour peu que le programmeur ou le graphiste soit aussi un peu *game-designer* dans l'âme) (voir glossaire). L'industrie du jeu vidéo est passée brusquement à des équipes énormes pour le développement d'un seul jeu, à des projets plus longs, et donc à des budgets en conséquence. Les méthodes de travail ont parfois un peu de mal à suivre cette évolution, ajoutant encore à la difficulté de mener à bien un développement de jeu massivement multi-joueurs. Il en est de même pour les outils de développement : même s'il est devenu naturel pour l'industrie de s'appuyer sur des bibliothèques graphiques ou comportementales pour développer un jeu-vidéo, l'utilisation d'outils logiciels dédiés à la réalisation de jeux massivement multi-joueurs n'est jamais envisagée. Une partie des difficultés dans la réalisation de ces applications est donc de l'ordre du génie logiciel, et nous abordons également ces aspects dans ce mémoire.

Le milieu industriel et la sphère académique

Il est particulièrement délicat de faire de la recherche concernant un domaine d'application si récent. En effet, les jeux massivement multi-joueurs sont des applications commerciales, et l'industrie du jeu vidéo a une longueur d'avance sur la recherche académique. C'est elle qui a une vue d'ensemble

de tous les problèmes à régler. Mais le monde industriel, de par sa nature concurrentielle, ne publie pas, et communique rarement de manière critique et objective ses résultats.

Il nous faut tempérer un peu ce jugement dans le cadre de l'industrie du jeu-vidéo : ce milieu communique plus sur les problèmes rencontrés que bien d'autres. Les développeurs de jeux-vidéo échangent tous les ans leurs expériences respectives à la *Game Developer Conference*, et des ouvrages collectifs rassemblent des contributions de professionnels sur les aspects techniques de la réalisation de jeux en ligne, comme par exemple la série des *Game Programming Gems*, dont le troisième volume notamment comportait une partie dédiée aux jeux en ligne [28], ou l'ouvrage *Massively Multiplayer Game Development*, complètement dédié aux techniques de développement des jeux massivement multi-joueurs [96]. La plupart des projets menés à terme donnent également lieu à la communication de *Post-Mortem*, donnant un retour d'expérience sur les problèmes rencontrés et les solutions retenues. Le Post-Mortem du jeu massivement multi-joueurs *Anarchy Online*, un des pionniers du genre, qui a essuyé quelques plâtres, s'est révélé très instructif pour les développements futurs [47], et suivant cet exemple, les développeurs de *Dark Age of Camelot* ont également fait partager leur expérience [41].

Mais dans une industrie aussi difficile que celle du jeu-vidéo, toujours sujette à des contraintes de temps et à des obligations de résultats rapides, on manque parfois de recul pour inventer des solutions plus génériques et à plus long terme, ou étudier les avancées de la recherche académique dans chacun des sous-domaines concernés, comme par exemple, le domaine de la recherche en informatique distribuée ou concurrente. Il est toujours difficile d'investir, dans le cadre d'un projet donné, du temps et des ressources en prévisions des éventuels projets futurs.

Nous pensons que la sphère académique doit prendre en charge la recherche à plus long terme dans le domaine d'application des jeux massivement multi-joueurs et commencer à communiquer clairement dans ce domaine. Elle a d'ailleurs commencé récemment à s'intéresser à ce domaine d'application, et le nombre de publications à ce sujet va grandissant depuis 2002.

Bien sûr, avant cela, le milieu académique s'est intéressé à des domaines d'application voisins, sujets à des difficultés que l'on retrouve dans les jeux massivement multi-joueurs, comme la diffusion d'informations à un grand nombre de machines au travers d'Internet, ou la synchronisation d'applica-

tions massivement distribuées. Même si les solutions proposées ne sont pas toujours utilisables directement pour réaliser des jeux massivement multi-joueurs, il est parfois possible de s'en inspirer pour répondre aux problèmes particuliers que soulèvent ces applications.

Une des principales difficultés de la réalisation de cette thèse a été de rassembler un état de l'art pertinent des techniques permettant de mettre au point des jeux massivement multi-joueurs. Tout d'abord car c'est un état de l'art qui doit à la fois regrouper le savoir-faire industriel et les quelques contributions académiques dédiées aux jeux massivement multi-joueurs. Ensuite parce qu'il a fallu passer en revue plusieurs domaines de recherche informatique connexes pour trouver les contributions pouvant être mises à profit dans le cadre de la réalisation de tels jeux. Enfin, au delà des techniques, il nous a fallu nous intéresser aux problématiques de génie logiciel, car encore une fois, les jeux massivement multi-joueurs sont des applications industrielles, et toute solution à la difficulté de leur mise au point ne saurait oublier de prendre en considération l'aspect pratique du développement de tels projets.

Contribution

L'objectif de cette thèse est de définir un cadre pour la mise au point des jeux massivement multi-joueurs, afin de simplifier, de rationaliser et de sécuriser leur développement.

Ce qui fait la singularité des jeux massivement multi-joueurs quand on les compare aux autres genres du jeu-vidéo, c'est la difficulté de mettre au point la manière dont le joueur interagit avec le monde virtuel dans lequel il évolue. Ces interactions sont de nature différentes : il peut s'agir d'un coup porté à un avatar, ou de la perception de son déplacement ; il peut s'agir de l'appropriation par son avatar d'un objet du monde virtuel, ou de transactions en monnaie locale ; il peut encore s'agir de communiquer avec les autres joueurs, au travers de messages texte, ou même audio.

Chaque type d'interaction a des besoins bien particuliers en terme de performances attendues et de caractéristiques techniques. Par exemple, une transaction entre avatars doit être sécurisée, il faut absolument que les parti-

cipants soient informés de son succès, et on ne peut pas se permettre que les paquets d'informations concernant cette transaction se perdent sur le réseau. A l'extrême inverse, il n'est pas nécessaire que lorsqu'un avatar se déplace, toutes les informations concernant ses positions successives soient transmises aux joueurs dont les avatars sont en mesure de le voir. Par contre il est important, dans le cadre de certains jeux, que ces informations arrivent avec la plus grande rapidité possible, surtout pour les jeux de combats ou le joueur doit pouvoir viser sa cible sans être handicapé par un retard de mise à jour de la position de cette dernière.

Nous montrons dans cette thèse que la définition des interactions dans un jeu massivement multi-joueurs est intrinsèquement liée aux spécifications fonctionnelles du jeu, le *game-play*. Le *game-play* (voir glossaire) est à la fois la description des règles du jeu, mais aussi la spécification de la manière dont chaque joueur interagit avec le monde virtuel dans lequel ce jeu se déroule. Il est donc unique et spécifique à chaque jeu massivement multi-joueurs.

A notre connaissance, il n'existe pas d'outil dédié à la conception de jeu massivement multi-joueurs prenant en compte cet état de fait, les solutions existantes ne permettant de mettre en œuvre qu'une variété limitée d'interactions, et ne permettant pas non plus de les modifier et de les tester facilement en regard du game-play désiré. Nous proposons donc l'idée d'un outil dédié à la mise au point des interactions, et décrivons comment cet outil peut être utilisé avec profit dans le cadre d'une méthodologie de développement appropriée.

Nous définissons ensuite le modèle sur lequel doit reposer un tel outil pour être générique et permettre la réalisation d'un maximum d'interactions.

Ce modèle est basé sur une modélisation très fine, en *états composés*, des différents objets qui composent le monde virtuel, sur chaque hôte de l'application. Les états contiennent les données définissant les objets auxquels ils correspondent. Pour chaque état, il est possible d'associer un ou plusieurs composants que nous appelons *réplications*. Ces réplications permettent de définir très finement quand, à qui et de quelle manière les mises à jour de l'état doivent être propagées.

Notre modèle comprend également un mécanisme de modification des données des états selon les événements se produisant dans l'application. Nous avons choisi pour ce mécanisme de mise à jour une sémantique simple et re-

productible, afin de privilégier une grande facilité d'utilisation, malgré l'aspect bas-niveau de notre solution.

Nous avons formalisé ce modèle, et nous en avons réalisé un prototype, sous forme d'un *framework* (voir glossaire), afin de montrer comment il peut s'utiliser pour modéliser quelques interactions typiques rencontrées dans des styles de jeux classiques et comment il peut former la base d'un outil d'intégration.

Organisation du mémoire

Il reste encore de nombreux non-adeptes des jeux massivement multi-joueurs. Nous allons donc commencer dans un premier chapitre, bref et imagé, par décrire ce qu'est un jeu massivement multi-joueurs, comment ce genre est né, et quelles sont les perspectives futures et évolutions probables de ce type de jeu. Nous y donnerons quelques exemples d'interactions typiques, en discutant quelles sont les propriétés qu'on en attend, suivant les *gameplay* classiques, afin de permettre au lecteur qui ne connaît pas le domaine d'application de mieux comprendre l'objectif de cette thèse.

Le deuxième chapitre est consacré aux difficultés techniques qu'on peut rencontrer lors de la définition des interactions d'un jeu massivement multi-joueurs. Nous décrivons en détail ces difficultés, y passons en revue les différentes solutions techniques actuellement utilisées pour les résoudre, et démontrons dans quelle mesure chaque solution permettant de satisfaire une caractéristique technique donnée a généralement un impact sur la qualité d'une autre caractéristique technique, résumant la définition d'une interaction à un compromis à équilibrer.

Le troisième chapitre est consacré à la présentation des problématiques qui ont alimenté notre réflexion dès le début de cette thèse. Nous y présentons des modèles de programmation concurrente, et y faisons un point sur les différentes méthodes de génie logiciel, afin de pouvoir ultérieurement montrer comment nos propositions peuvent s'intégrer dans un cadre réaliste et applicable à un développement industriel. Nous présenterons également ce qui a été le déclencheur de cette thèse, la technologie Scol, développée par la défunte société de développement de jeux en ligne Cryo-Networks, ancien

employeur de l'auteur.

Nous montrons dans le quatrième chapitre qu'il ne peut pas y avoir de solution générique à tous les problèmes rencontrés dans la réalisation d'un jeu massivement multi-joueurs. Nous décrivons à partir de cette constatation la réflexion qui nous a amenée à la conception du modèle présenté dans cette thèse. Nous en décrivons les concepts principaux, et l'esprit général, basé sur une modélisation très fine en états du jeu et de la manière dont leur mise à jour s'effectue le long de l'application distribuée qu'est un jeu massivement multi-joueurs. Nous le situons pour conclure par rapport à des modèles récents et voisins.

Le cinquième chapitre décrit en détail le modèle que nous avons conçu au cours de cette thèse. Nous donnons une description formelle du comportement du modèle, puis quelques exemples d'utilisations classiques pour des situations fréquemment rencontrées dans le cadre de son utilisation.

Le sixième chapitre présente le *Fill-In-The-Gaps Toolkit*, le prototype que nous avons développé en Java pour implémenter notre modèle. Ce prototype est destiné à démontrer comment le modèle peut s'intégrer dans un environnement de développement. Il se termine sur un exemple simple d'utilisation de notre *framework* pour le développement d'une petite application, dont le code plus complet est disponible en annexe.

Nous concluons par une synthèse du travail réalisé, et sur les perspectives ouvertes par notre solution concernant la conception d'un environnement de développement dédié à la réalisation d'un jeu massivement multi-joueurs.

Le lecteur peut également trouver un glossaire des termes que nous avons jugés intraduisibles ou propres au domaine d'application à la fin de ce mémoire. La première occurrence de chaque mot du glossaire est signalée, comme vous avez sans doute déjà pu le constater pendant la lecture de cette introduction.

Chapitre 1

Brève histoire des jeux multi-joueurs

Nous allons raconter ici une brève histoire des jeux massivement multi-joueurs, exercice subjectif s'il en est : il n'est pas question de passer en revue la totalité de la production vidéo-ludique des 30 dernières années, et le critère qui a valu aux jeux sous-cités d'être évoqués dans ce chapitre n'est pas (toujours) leur succès. L'auteur a *essayé* de rester objectif, et de présenter les jeux qui présentent une rupture technologique ou qui sont significatifs d'une autre évolution marquante dans le secteur des jeux-vidéo. Il est toutefois fortement probable qu'entre deux jeux de mêmes qualités, l'auteur se soit laissé influencer par ses goûts ou ceux de son entourage proche. Pour une histoire plus complète et détaillée, nous invitons le lecteur à se référer au livre de Stéphane Natkin [75]

1.1 Les Multi-User Dungeons

Au début, il y a le *Multi-User Dungeon*...

Non, reprenons. Au tout tout début, il y a eu les jeux de rôles sur table. Parce que oui, avant, on jouait à plusieurs sans avoir besoin d'Internet et d'un ordinateur.

Un jeu de rôles sur table propose aux participants d'incarner un person-

nage donné tout au long d'une histoire qui évolue selon les décisions de ces joueurs et des jets de dés. Un maître de jeu organise le déroulement, c'est lui qui est le maître de la trame de l'histoire et le garant du respect des règles.

Après les premiers jeux d'aventures sur ordinateurs, nés dans le milieu des années 70, le premier jeu de rôles multi-joueurs sur ordinateur naît en 1978, à l'université de l'Essex en Angleterre. MUD (pour Multi-User Dungeon) est développé par Roy Trubshaw et Richard Bartle et connaît très vite un vif succès.

C'est un jeu de rôles, comme ceux qu'on joue sur papier, mais dont le maître de jeu est un programme. Les joueurs, connectés à l'application, incarnent des personnages dans un monde virtuel qui leur est présenté par du texte. Les quelques professions que peuvent exercer les joueurs sont complémentaires les unes des autres, ceci afin de construire un univers dans lequel les joueurs auront besoin les uns des autres pour évoluer.

Le principal problème était alors l'équilibrage du jeu lui-même : en termes de règles du jeu, pour que les différentes professions soient également compétentes, mais aussi de manière plus surprenante en terme de natures de joueurs. En effet, lorsqu'on joue à un jeu de rôles sur table, on connaît en général tous les participants, et surtout, on est en leur présence physique, ce qui incite à un certain fair-play, normalement arbitré par la personne physique qu'est le maître du jeu. Lorsqu'on joue sur son ordinateur, camouflé par le réseau informatique, et qu'on n'est plus en présence physique des autres joueurs potentiellement inconnus, on a parfois tendance à oublier les bonnes manières.

Ainsi, MUD et ses successeurs (appelés désormais MUDs, cet acronyme étant devenu le nom du genre), même s'ils n'ont plus grand chose à voir en terme de technique avec les jeux massivement multi-joueurs actuels, ont fourni des laboratoires d'expérimentation pour les *game-designs* (voir glossaire) et représentent toujours l'essence même du jeu massivement multi-joueurs. Les problèmes d'équilibrage qui se posent sont toujours les mêmes, et certaines études faites très tôt dans le cadre de ces jeux restent toujours d'une criante actualité, comme le fameux *Clubs, Hearts, Diamonds and Spades* de Richard Bartle sur le comportement et les relations qu'entretiennent les différentes typologies de joueurs dans un monde virtuel [5].

Un grand nombre de MUDs sont actuellement encore en activité sur In-

ternet. La figure 1.1 présente une interface moderne permettant de jouer à un MUD de manière un peu plus élaborée qu'avec un simple terminal en ligne de commande[1].

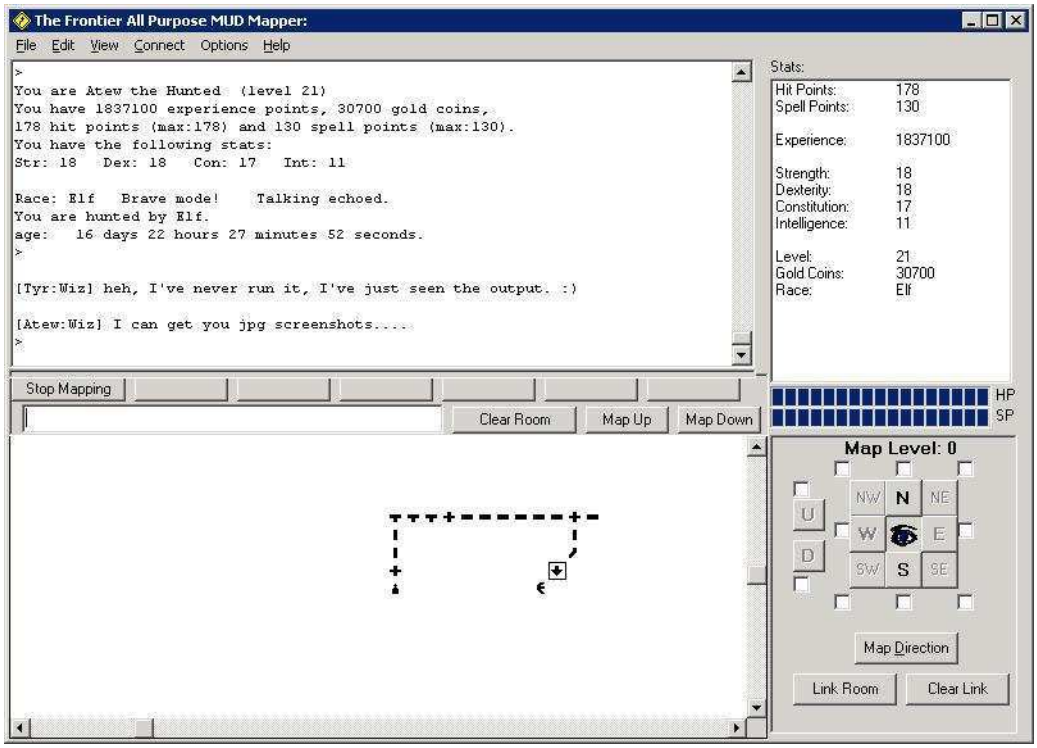


FIG. 1.1 – Un client moderne de Multi-User Dungeon (Frontier MUD Mapper par Veramacor)

1.2 Evolution vers les jeux de rôles persistants

Ultima Online[2] naît en 1997. C'est un des tout premiers jeu massivement multi-joueurs sur Internet. Comme MUD, c'est un jeu d'aventure multi-joueurs. Il se déroule dans un monde persistant d'inspiration médiévale-fantastique. Il consiste en plusieurs mondes indépendants, dans lesquels co-

[1] http ://frontier.mudservices.com
[2] Ultima Online, de la société Origin Systems : `http ://www.uo.com/`

habitent simultanément des centaines de joueurs. Comme MUD, ce jeu se heurte à des problèmes d'équilibrage. C'est un jeu qui permet les combats entre joueurs et où il vaut mieux essayer d'éviter les mauvaises rencontres.

Le joueur paie désormais un abonnement pour se connecter à l'univers du jeu, ce qui le rend d'autant plus sensible aux problèmes techniques de ce nouveau genre : Ultima Online essuie les plâtres.

La figure 1.2, tirée des aventures de B0N3D00D et pLaTeDeWd [3] illustre les problèmes récurrents de synchronisation au sein du jeu. Malgré ces pro-

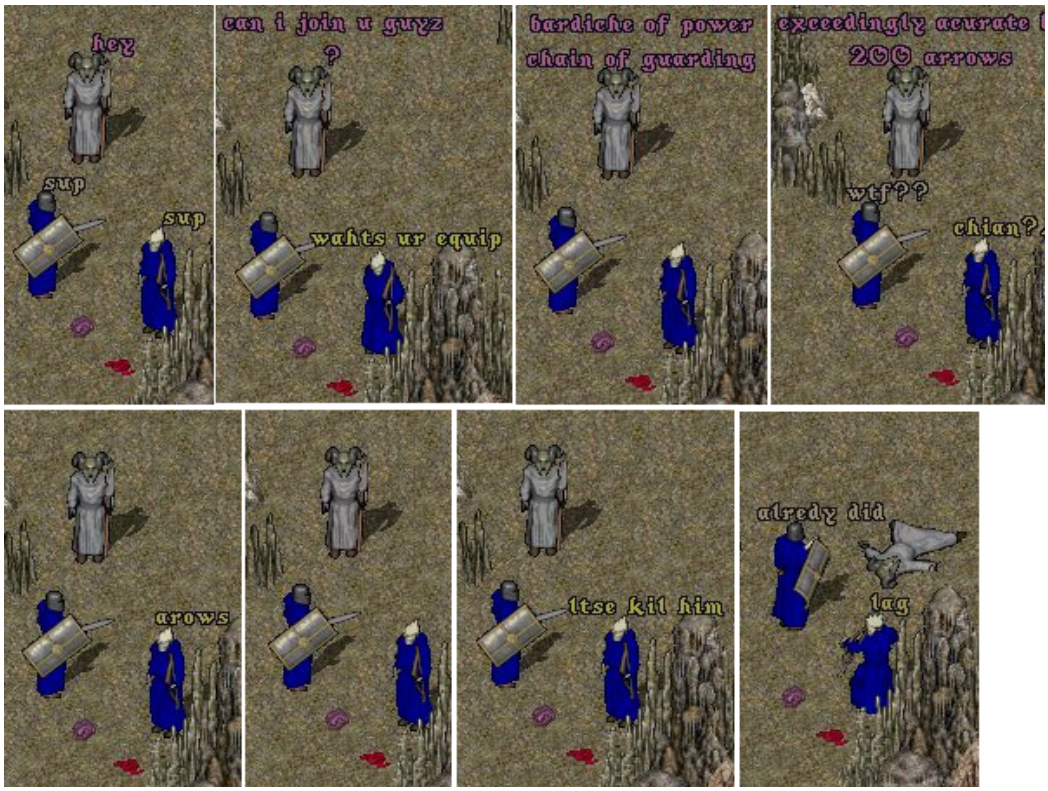


FIG. 1.2 – La synchronisation dans Ultima Online (Origin Systems/Electronic Arts, 1997)

blèmes techniques, ce jeu a rencontré un grand succès et il a fallu attendre

[3] Les aventures de B0N3D00D et pLaTeDeWd dans Ultima Online : `http ://www.iqto.com/bp/oldstuff.htm`

Everquest, sorti en 1999, pour trouver un jeu massivement multi-joueurs qui le batte en popularité et nombre d'abonnés.

En l'espace de deux ans, entre la naissance de ces deux jeux, les connexions à haut débit sont apparues et les ordinateurs personnels se sont également considérablement améliorés. Everquest [34] (figure 1.3) est un jeu qui dispose d'un rendu en trois dimensions, et se déroule dans un univers à la Tolkien. Les problèmes techniques sont moindres. Les serveurs d'Everquest sont capables de supporter jusqu'à 3000 joueurs simultanés.

Cependant, les combats sont toujours basés sur du tour par tour assez peu réaliste, manquant de réactivité, et la puissance des coups portés à l'adversaire se calcule encore avec un facteur de hasard, comme au bon vieux temps des dés lors des parties de jeux de rôles sur table.

FIG. 1.3 – Everquest (Sony Online Entertainment, 1999)

1.3 La révolution des jeux d'action à la première personne

Pendant que les joueurs *old-school* amateurs de jeux de rôles sur papier découvrent Everquest et s'investissent dans des mondes virtuels persistants, la jeune génération des joueurs prend l'habitude de s'entretuer dans des jeux multi-joueurs communément appelés *First Person Shooter* (FPS). Ce style de jeu émerge au début des années 90, à partir du moment où les ordinateurs personnels commencent à gérer correctement les rendus en trois dimensions. On peut notamment citer, comme exemples emblématiques, les titres Quake [84] (Figure 1.4) et Half-Life [51] dont les versions successives ont remporté un grand succès.

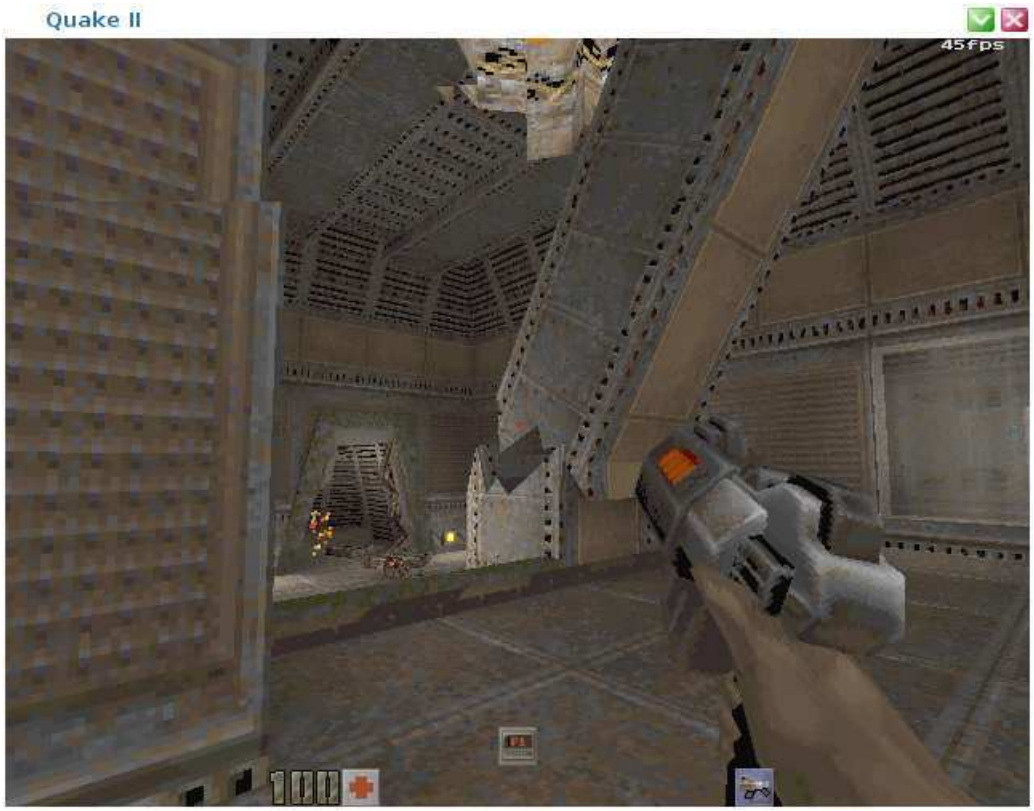


FIG. 1.4 – Quake II (id Software/Activision, 1997)

Dans ces jeux, le joueur est immergé en vue subjective dans son propre avatar, et voit sur son écran la représentation du monde virtuel dans lequel se déroule la partie comme à travers les yeux de cet avatar. Il s'agit en règle générale de s'affronter en équipe ou contre des *Personnages Non-Joueurs* (voir glossaire) pour des objectifs précis à atteindre, à l'aide d'un vaste choix d'armes militaires aux qualités létales diverses et variées.

Tout d'abord prévus pour être joués en solitaire, ces jeux ont très vite proposé des versions multi-joueurs, pour jouer entre amis sur réseau local, et leur popularité est devenue telle qu'ils donnent lieu à de multiples conventions et championnats partout dans le monde. Ceci est du en partie au fait que les qualités «sportives» des joueurs ont une grande influence sur leurs résultats. En effet, ces jeux sont aux parties de paint-ball ce que les jeux comme Everquest sont aux jeux de rôles sur table : les déplacements du joueur, ses réflexes, sa capacité à viser correctement, les stratégies d'équipe et l'appréciation globale des actions en cours sont les qualités qui font la différence.

Au contraire des jeux massivement multi-joueurs classiques, les FPS ne se jouent pas dans des univers persistants mais par parties (sessions), sur réseau local, puis lorsqu'Internet s'est démocratisé, sur des serveurs en ligne gérant plusieurs sessions simultanées. Le joueur n'achète que le logiciel du jeu, et ne paie pas d'abonnement mensuel.

1.4 Evolution récente

Les jeunes joueurs de FPS ont désormais grandi et disposent d'une carte de crédit qui leur permet de jouer à des jeux massivement multi-joueurs. Avec ce saut de génération, les *game-design* ont essayé d'évoluer au rythme de leurs potentiels clients et de fournir des styles de jeux hybrides, des jeux massivement multi-joueurs dans lesquels on se déplace et se bagarre au même rythme que dans un FPS.

La motivation principale de l'industrie du jeu-vidéo dans le développement de nouveaux types de jeux plus dynamiques et plus rapides vient aussi de la volonté d'attirer une nouvelle cible de consommateurs, peu désireux de devoir passer tout leur temps libre dans le jeu pour pouvoir en profiter plei-

nement. Neocron [76] (figure 1.5) est un exemple d'un tel jeu massivement

FIG. 1.5 – Neocron (Reakktor/CDV Software Entertainment, 2002)

multi-joueurs, qui propose le même genre de classes d'avatars qu'un jeu massivement multi-joueurs traditionnel, et qui innove également en proposant un univers post-apocalyptique cyberpunk. Le façon de s'y déplacer, de s'y bagarrer, ressemble beaucoup plus à celle d'un FPS qu'à celle d'Everquest, mais l'univers est moins vaste, et plutôt taillé pour des centaines que pour des milliers de joueurs.

Cependant, le jeu a souffert de gros défauts techniques et les serveurs ne sont plus maintenus, après seulement deux années d'exploitation. Neocron, comme Ultima Online en son temps, a essuyé les plâtres.

Les jeux massivement multi-joueurs sortis les plus récemment reviennent aux bonnes vieilles recettes qui ont toujours marché. Le récent World of

Warcraft (figure 1.6), sans être un clône d'Everquest, reprend le *game-play* traditionnel et désormais sans risque du genre. Les cibles, une fois sélectionnées, ne seront touchées qu'en fonction du facteur chance du jeu de dé et pas en fonction de l'adresse au tir de l'attaquant, tandis que la cible à beau courir dans tous les sens, elle reste dans son collimateur. Moins de dynamisme, moins de réactivité, mais aussi moins de problèmes techniques. Les jeux massivement multi-joueurs actuels se ressemblent tous beaucoup.

FIG. 1.6 – World Of Warcraft (Blizzard Entertainment/Vivendi Universal, 2004)

1.5 Synthèse

Selon le style de *game-play*, les interactions diffèrent donc énormément d'un jeu à l'autre.

Récapitulons par exemple les différentes manières de modéliser l'interaction qui consiste à tirer sur un joueur.

Dans un FPS, cette interaction est modélisée de façon à être la plus réaliste possible : le joueur doit viser un avatar en mouvement, et la cible ne sera touchée que s'il a été assez habile. La perception du déplacement d'un avatar représentant un joueur distant par chaque participant doit donc être la plus fiable possible. Afin de réaliser cet objectif, les mises à jour de la position de la cible selon les actions du joueur adversaire doivent donc arriver le plus vite possible. On utilise alors en général un protocole réseau minimisant les temps de transfert, au détriment de l'assurance que tous les messages arriveront à destination, et on envoie les mises à jour en boucle le plus rapidement possible.

Dans un jeu massivement multi-joueurs comme Everquest, les positions des joueurs n'ont pas tant d'importance. Une fois la cible sélectionnée, par simple clic du joueur, il suffit de lancer une ou plusieurs attaques, et c'est le serveur de jeu qui décidera, entre hasard et niveau de compétences des joueurs impliqués, si la cible est touchée et l'étendue des dommages. Il n'est donc pas nécessaire de transmettre les positions des déplacement des joueurs avec la même rapidité que dans le cas précédent. En fait, dans le cas de ce jeu, c'est même déconseillé : la zone dans laquelle se déroule le combat peut être très peuplée, et les machines des joueurs risquent d'être saturées de messages qu'elles n'ont pas le temps de traiter.

Le but de cette thèse est de proposer une solution pour aider à la mise au point de jeux utilisant des *game-play* différents et originaux, sans pour autant prendre trop de risque dans le déroulement du projet : si les sociétés sont si frileuses à innover, c'est que la réalisation d'un jeu massivement multi-joueurs est un projet énorme, complexe et coûteux, comme nous allons le voir dans la suite de ce manuscrit, et qu'il manque d'outils pour aider à la réalisation de jeux innovants.

Chapitre 2

Besoin de performances : revue technique

Pour mieux comprendre le défi que représente la réalisation d'un jeu massivement multi-joueurs, nous allons décrire dans cette partie les enjeux techniques à prendre en considération. Nous allons également passer en revue les diverses techniques actuellement utilisées pour essayer de répondre aux contraintes très fortes auxquelles sont sujettes ces applications.

Le milieu académique a tardé à s'intéresser au domaine des jeux en ligne sur Internet. Jusqu'à très récemment, les travaux exploitables dans le cadre des jeux massivement multi-joueurs étaient principalement des communications issues du monde industriel. La plupart des travaux antérieurs à l'année 2000 avaient pour objectif principal les applications de simulation militaire en réseau local, ou ne s'intéressaient aux jeux multi-joueurs qu'en tant qu'application de démonstration pour des travaux concernant l'optimisation de protocoles réseau, ou de synchronisation d'application distribuée. Un grand nombre des travaux dans ces domaines sont applicables au domaine des jeux massivement multi-joueurs : il y a une intersection commune avec toutes les problématiques rencontrées dans le développement des types d'applications pré-citées. Une bonne synthèse des travaux antérieurs à 2000 concernant les mondes virtuels peut être trouvée dans le livre écrit par Sandeep Singhal et Michael Zyda [93].

Le problème avec les jeux massivement multi-joueurs commerciaux, c'est

qu'ils se frottent à toutes ces problématiques en même temps. Beaucoup de solutions développées dans le cadre de la simulation militaire ou de la réalité virtuelle pour une problématique donnée ne sont pas satisfaisantes car elles empêchent d'en régler une autre. Publiée en 2002, une étude [59] faite par Jouni Smed, Timo Kaukoranta et Harri Hakonen tente d'établir un lien entre la recherche académique et les problèmes rencontrés par les industriels réalisant les jeux multi-joueurs. Les auteurs y définissent les points communs entre les caractéristiques des applications de simulations traitées dans l'académique et celles des jeux multi-joueurs sur Internet. Ils définissent les problématiques spécifiques à ce dernier domaine d'application, dont les innovations sont à l'époque principalement issues du monde industriel.

Dans ce chapitre, nous allons commencer par décrire toutes les difficultés et contraintes techniques que l'on rencontre dans le cadre de la réalisation d'un jeu massivement multi-joueurs.

Après avoir identifié lesquelles de ces difficultés sont en rapport avec la définition des interactions du *game-play*, nous ferons le tour des solutions permettant de jongler entre les différents impératifs de performances : la force des contraintes compliquant la réalisation d'un jeu massivement multi-joueurs dépend en grande partie du style de jeu que l'on veut réaliser. Suivant le *game-play*, certaines peuvent prendre plus d'importance que d'autres, et on peut parfois négliger la qualité d'une des caractéristiques techniques au profit d'une autre. Pour un état de l'art plus général, non centré comme ici sur la mise au point des interactions du *game-play* et abordant d'autres difficultés techniques comme l'utilisation du son et la mobilité, on peut se référer à [74].

2.1 Caractéristiques techniques d'un jeu multi-joueurs

Dans cette partie, nous allons définir ce qu'est un jeu massivement multi-joueurs en termes techniques, en donnant une liste de ce qui caractérise ces applications distribuées sur Internet.

2.1.1 Persistance

Les jeux massivement multi-joueurs sont des applications persistantes. Cela signifie que le monde virtuel est accessible vingt-quatre heure sur vingt-quatre et sept jours sur sept. Une fois que le jeu est lancé, la partie ne s'arrête pas, et les joueurs peuvent rejoindre à tout moment cet univers qui évolue en permanence selon leurs actions.

En réalité, la plupart des jeux multi-joueurs prévoient des maintenances régulières ou sporadiques, lors desquelles les serveurs sont arrêtés, afin de rajouter du contenu au jeu, ou de réparer des bugs.[1] Mais la plupart des modifications de contenu simples, comme le changement du comportement d'un personnage non-joueur peuvent souvent s'effectuer sans avoir à arrêter l'application. Par exemple, lorsque les concepteurs du jeu ont prévu cette possibilité à l'avance en utilisant des dispositifs de modification dynamique des scripts permettant de contrôler ces comportements.

Il est particulièrement important de disposer d'une bonne politique de sauvegarde de l'état du monde : lors du re-démarrage de l'application, le jeu doit se retrouver dans le même état que lorsqu'il s'est arrêté. Comme l'arrêt des serveurs peut ne pas être volontaire, mais conséquence d'un problème technique quelconque, il est également important de bien gérer la sauvegarde périodique des éléments les plus importants de l'état du monde virtuel.

Afin de garantir le meilleur service possible, il est indispensable que l'application soit facile à maintenir et à faire évoluer. Un jeu massivement multi-joueurs est exploité pendant des années et le coût de maintenance d'un jeu mal fait peut réduire grandement les bénéfices.

Le principal risque si la persistance de l'application est mal gérée est le *rollback* (voir glossaire) : en cas de corruption de l'état du jeu, pour des raisons techniques ou du fait d'une intervention extérieure malveillante exploitant une faille de l'application, il est parfois nécessaire de devoir revenir à un état antérieur, dont la sauvegarde est intacte. Le rollback est craint à la fois par les joueurs, qui risquent ainsi de perdre de nombreuses heures de jeu (et donc toute leur progression dans le monde, en termes de compétences, d'acquisition d'objets et d'argent virtuel), et par les sociétés exploitant les

[1] C'est le fameux «*Patch day don't play*» d'Everquest, dont les opérations de maintenance duraient fréquemment plus longtemps que prévu.

jeux, qui n'aiment pas fâcher leurs clients qui prennent souvent leur vie virtuelle très à cœur.

2.1.2 Synchronisation dans une application distribuée

Afin de pouvoir proposer une bonne immersion aux joueurs, l'état global du jeu doit être partagé par tous les hôtes de l'application à tout instant : dans un monde idéal, chaque joueur devrait voir en permanence sur son ordinateur les données représentant l'état actuel du jeu et partager cet état avec tous les autres joueurs.

Cette description est un paradoxe en elle-même : chaque changement du monde sur un des hôtes de la distribution doit être répercuté par le réseau sur les autres hôtes de la distribution. La propagation d'une information par un réseau prend du temps, même si elle peut être rapide à l'échelle humaine. Les informations ont également un certain volume, qui peut devenir handicapant dans certaines conditions de débit, et Internet, en tant que réseau hétérogène et au comportement souvent imprévisible, n'arrange pas les choses.

Ceci nous mène directement au problème de la consistance de l'accès aux données dans une application distribuée, c'est-à-dire de la possibilité d'avoir un état du jeu commun entre tous les hôtes de l'application énoncé dans [93] : *Il est impossible d'autoriser l'état dynamique partagé à changer fréquemment tout en garantissant que tous les hôtes d'une application accèdent simultanément à des versions identiques du monde virtuel.*

Il faut donc être capable de faire en sorte que ce problème de synchronisation ne soit pas perceptible par les joueurs, ce qui peut être particulièrement délicat, en particulier lorsqu'il s'agit de jeux où la réactivité du joueur a une grande importance : par exemple dans un FPS, lorsque le décalage dépasse le temps du réflexe humain, il devient très vite difficile de jouer dans de bonnes conditions.

2.1.3 Sécurité

Dans un jeu massivement multi-joueurs, le problème de la sécurité est crucial. Les modèles commerciaux, axés sur un abonnement pour l'accès au

jeu, font qu'on y retrouve tous les problèmes communs aux applications de services sur Internet. Sécurisation de l'authentification du joueur, afin qu'un autre ne puisse accéder à ses données et avatars personnels, paiement sécurisé, résistance des serveurs aux attaques réseau classiques, sont les caractéristiques communes aux applications de services Internet et aux jeux massivement multi-joueurs.

Aux problèmes de sécurité classique s'ajoutent les conséquences liées à la triche : si le jeu laisse la porte ouverte aux tricheurs, les joueurs honnêtes partiront très vite, frustrés, et c'est la fin du monde (virtuel).

En effet, comme un jeu en ligne se déroule dans un monde persistant, où l'économie et l'évolution des personnages selon les actions des joueurs sont soigneusement, délicatement mises au point, l'équilibre est fragile. Un joueur qui découvrirait un moyen de faire de l'argent facile alors que ça n'est pas prévu par les concepteurs peut ruiner l'économie du monde virtuel : l'argent circule, inonde le jeu, et finalement ne signifie plus rien. La moindre faille dans le jeu sera fatalement exploitée par les joueurs.

Un déséquilibre du monde virtuel a bien plus de conséquences dans le cadre d'un jeu persistant que dans un jeu multi-joueurs qui se joue par sessions. En effet, la seule manière de le rééquilibrer, une fois la faille de sécurité corrigée, sera souvent de revenir à un état antérieur non corrompu du jeu en effectuant un *rollback*.

Il est assez délicat de trancher sur les cas relevant de la triche ou non dans les jeux massivement multi-joueurs. Par exemple, dans certains jeux offrant un style de combat «réaliste» en ville, la technique qui consiste pour un joueur à se dissimuler derrière un élément du décor en attendant de pouvoir tirer dans le dos des cibles faciles sans prendre de risques grâce à une définition approximative des collisions de cet élément peut être indifféremment considérée selon les jeux comme une tactique maligne, un manque de fair-play, ou un abus d'une faille dans la construction du monde.

Jianxin Jeff Yan et Hyun-Jin Choi ont défini une taxinomie de la triche dans les jeux massivement multi-joueurs [103] que nous allons discuter dans cette partie. En effet, quelques uns des types de triche retenus par les auteurs nous semblent relever du cas par cas. D'autres nous semblent plutôt relever du *grief-play* (voir glossaire), qui peut être défini comme le fait d'adopter un style de jeu autorisé par le *game-design*, mais à en exploiter les possibilités

pour gâcher l'expérience des autres joueurs. On peut trouver une taxinomie du *grief-play* pour des jeux comme Everquest dans [42]. Nous nous appuierons également sur les exemples de fraudes constatées dans [23]. Les montants impliqués dans certaines fraudes concernant les jeux massivement multi-joueurs évoquées dans cette étude atteignent plusieurs milliers de dollars, et les auteurs évoquent une moyenne de 200 fraudes criminelles recensées par jour.

2.1.3.1 Tricher à l'aide d'un complice :

Yan et Choi donnent comme exemple le cas du jeu de bridge, où deux joueurs complices (ou un même joueur incarnant deux avatars distincts) peuvent connaître leurs mains respectives sans que les autres s'en rendent compte. Nous ajoutons à cette catégorie un exemple qui nous paraît plus représentatif des jeux massivement multi-joueurs : la technique du *power levelling* (voir glossaire) consiste, dans un jeu massivement multi-joueurs, à utiliser les capacités d'un avatar puissant pour faire progresser beaucoup plus rapidement qu'à la vitesse prévue par les concepteurs un avatar plus faible. Par exemple, l'aider à obtenir des objets qu'il n'aurait pas pu acquérir seul avant plusieurs dizaines d'heures de jeu, partager en faisant équipe avec lui l'expérience acquise en combattant un personnage non joueur qu'il n'a aucune chance de vaincre seul. Cela peut aussi être, pour les jeux incluant une notion de prestige dans l'habilité au combat entre joueurs, laisser un avatar se faire tuer un grand nombre de fois par un autre afin d'acquérir sans risque les galons de la gloire.

Le *power levelling* est possible dans beaucoup de jeux, et n'est souvent pas considéré comme de la triche, même s'il peut être ressenti comme une injustice ou un manque de fair-play par les joueurs qui n'en profitent pas.

2.1.3.2 Tricher en abusant des mécanismes du jeu :

Un exemple très commun de ce type de triche, donné par Yan et Choi, est la fuite : si quelque chose commence à mal tourner dans le jeu, que le joueur risque de perdre de l'expérience ou des possessions virtuelles car il n'a plus le dessus, il se déconnecte brutalement et disparaît du monde virtuel. L'action n'étant pas allée jusqu'à son terme, le joueur sauve son avatar par des moyens qui ne font pas partie des mécanismes du *game-play*. Nous ajoutons à cette

catégorie les phénomènes de *kill steal* (voir glossaire), consistant par exemple dans certains jeux à attendre qu'un joueur aie presque fini d'achever un personnage non-joueur pour donner le coup fatal afin d'emporter l'expérience ou les récompenses associées, et le *loot steal* (voir glossaire), consistant dans certains jeux à ramasser très vite les récompenses associées au succès d'un joueur lors d'un combat, avant que ce dernier n'ait eu le temps de le faire.

Comme la triche par complicité, la perception de ces comportements n'est souvent pas considérée comme de la triche, au pire comme un comportement déviant qui ne vaudra pas à ses adeptes une immense popularité dans le monde virtuel. Ils sont plus souvent considérés comme le problème du *player killer* (voir glossaire), ces joueurs dont la principale motivation est de ruiner le plaisir de jeu des autres joueurs : des comportements nuisibles au plaisir de la plupart des joueurs, mais légitimes. Nous pensons qu'ils rentrent plutôt dans la catégorie du *grief-play* que de la triche à proprement parler.

2.1.3.3 Triche liée au marchandage entre joueurs d'objets virtuels

Certains objets virtuels d'un jeu massivement multi-joueurs sont extrêmement rares et difficiles à obtenir. Or, à cause de l'importance que prend parfois pour les joueurs leur course au prestige, avatars et objets virtuels se vendent aux plus offrants à des prix jugés indécents par le commun des mortels sur des services d'enchère en ligne. Les abus de confiance entre les joueurs peuvent donc être très mal ressentis par les joueurs floués, lorsque les jeux ne permettent pas de contrôler les échanges.

Bien sur, il ne s'agit pas toujours d'argent réel qui est en jeu, mais parfois tout simplement des escroqueries virtuelles entre joueurs. Là encore, malgré la frustration du joueur virtuellement escroqué, la définition de ce comportement comme étant de la triche n'est pas tout à fait claire et nous semble plutôt relever du *grief-play*.

2.1.3.4 Extorsion frauduleuse des identifiants de connexion des joueurs

Là encore, la principale motivation de ce type de triche est la possession des objets du monde virtuel. Dans [23], les auteurs répertorient un certains

nombre de pratiques frauduleuses voire criminelles visant à l'obtention des paramètres de connexion d'un joueur, allant de l'agression physique et du kidnapping à la classique utilisation d'un virus troyen (ce type de programme s'installe sur un ordinateur comme les virus habituels et envoie des informations personnelles par le réseau à une personne qu'on peut alors supposer malintentionnée), en passant par les attaques «dictionnaire» (il s'agit également d'attaques classiques, basées sur le fait que les utilisateurs répugnent à utiliser des mots de passe compliqués, et qui consistent à essayer une liste de mots et de leurs variantes évidentes, inversées ou en jouant avec les lettres majuscules et minuscules).

Ici, plutôt que de la triche, les pratiques relèvent du pénal et des problèmes de sécurité réseau classique.

2.1.3.5 Escroqueries relevant de l'abus de confiance pour l'obtention des identifiants de connexion

Les habitués des services de paiement en ligne ou des sites marchands ont tous reçu, un jour ou l'autre, des e-mails les incitant à envoyer leurs identifiants pour l'accès sécurisé à ces services par retour de courrier sous un prétexte technique quelconque et fallacieux. Ce type d'abus de confiance est bien évidemment applicable aux jeux massivement multi-joueurs.

Ces méthodes externes au jeu lui-même sont clairement à classer du côté des pratiques frauduleuses et non de la triche, si on oublie ce simple fait fréquent : les joueurs ont souvent tendance à révéler eux même en toute confiance et sans incitation leurs identifiants de connexion à leurs camarades de jeux car certains avatars ont des spécialités indispensables au jeu en équipe, et il est pratique pour les autres membres de l'équipe de pouvoir disposer de cet avatar à volonté. Les abus de confiance se produisent plus fréquemment lorsqu'on se croit protégé par l'anonymat que procurent les mondes virtuels.

2.1.3.6 Triche à l'aide d'attaques *Déni de Service* contre les autres joueurs

Les attaques réseau de type *Déni de Service* sont bien connues dans le domaine de la sécurité sur Internet.

Une telle attaque peut tout simplement être réalisée après l'obtention des identifiants de connexion d'un joueur par les méthodes décrites au paragraphe précédent, l'empêchant ainsi de se connecter lui-même au jeu. Bloquer ainsi un joueur indispensable à une équipe contre laquelle on est en conflit dans le cadre du jeu peut suffire à déséquilibrer injustement le jeu.

Les techniques plus classiques d'attaques de ce type, qui consistent à saturer la connexion d'un joueur en lui envoyant des messages, pour ralentir sa connexion et ainsi avoir un avantage certain quand une bonne réactivité du joueur est nécessaire, par exemple lors de combats dans un FPS. L'adversaire aura beau avoir des réflexes, si soudainement sa connexion réseau est ralentie, il sera très handicapé. Ce type d'attaque implique néanmoins que le tricheur puisse connaître l'adresse IP de son adversaire.

2.1.3.7 Triche due au manque de confidentialité

Lorsque les commandes sont envoyées en clair par le réseau des machines des joueurs aux serveurs, une autre attaque courante consiste à analyser les messages transitant sur le réseau, et à les modifier au passage ou à en insérer d'autres. L'obtention des identifiants de connexion est bien sûr sujette à cette attaque, mais aussi tout ce qui concerne les actions des joueurs dans le monde virtuel, afin d'influer et de modifier l'état du monde pour en gagner un avantage.

2.1.3.8 Triche due à un manque d'authentification

Nous avons évoqué plus haut les problèmes liés à l'authentification des joueurs par leurs identifiants de connexion, mais il est aussi important que le serveur du jeu puisse être authentifié par les logiciels clients utilisés par ces derniers pour se connecter : en effet, pour peu que quelqu'un réussisse à orienter les joueurs vers un serveur modifié à cet effet, il peut collecter les

identifiants de connexion des participants.

2.1.3.9 Problèmes de sécurité internes aux sociétés

Toujours dans [103], les auteurs définissent une catégorie à part entière provenant des risques associés aux employés des sociétés exploitant les jeux : en effet, il est souvent prévu un mode de super-utilisateur, associé à des droits particuliers, comme la création d'objets virtuels rares et convoités, et les auteurs citent des cas de fraudes réelles constatées.

2.1.3.10 Triche par modification du logiciel utilisé par les joueurs, ou par modification des données du jeu

Ce type de triche n'est pas nouveau, puisque la modification des programmes de jeu traditionnels pour pouvoir les finir plus facilement est une chose courante. Mais alors que dans les jeux solitaires, la triche ne nuit après tout qu'au tricheur qui réduit ainsi la durée de vie de sa partie pour le simple plaisir de battre l'ordinateur, la modification du programme nuit désormais aux autres joueur : les premiers *aimbots* (voir glossaire) pour le jeu Quake permettaient à l'utilisateur de toujours viser juste.

Voici un autre exemple très simple de ce type de triche : le joueur modifie son programme client, afin que tous les avatars représentant les autres joueurs lui soient désormais affichés en costume d'oursin bardé de piques très longues dans tous les sens. Désormais, il lui sera très facile d'échapper à ses opposants, cachés en embuscade au coin d'une rue derrière un mur, puisque les piques du costume, passant à travers les autres éléments du décor, les signaleront.

Même si ce type de triche est plus complexe à réaliser et nécessite un minimum de compétence technique, à partir du moment où le logiciel modifié existe, il est à parier qu'il s'écoulera très peu de temps avant qu'il soit disponible à beaucoup de joueurs, tant il est vrai qu'il est plus valorisant encore pour les tricheurs d'expliquer la manière très intelligente avec laquelle ils ont procédé que de garder le secret de leurs performances pour eux.

2.1.3.11 Exploitation de bugs, ou de faiblesses de conception

Les auteurs de [103] font de l'exploitation des faiblesses du jeu une catégorie de triche à part entière. Il nous semble cependant qu'il y a une intersection commune avec bien des catégories précédemment citées. Par exemple, si une personne est capable de récupérer par simple espionnage des paquets envoyés au travers du réseau parce que les mots de passe ne sont pas encodés, ou pas de manière assez sûre, ça n'est pas seulement un manque de confidentialité (voir partie 2.1.3.7, page 43) mais une faiblesse évidente de conception.

De même, la plupart des triches utilisant une version modifiée du logiciel client se basent sur une de ses faiblesses, comme par exemple certaines techniques de masquage de la latence qui rendent trop facilement détectables sur la machine du joueur à l'instant t des données qu'il n'est censé exploiter qu'à l'instant $t + 1$.

Insister sur la nécessité d'avoir une application sûre, vérifiée et de qualité n'est néanmoins pas inutile dans la mesure où les conséquences peuvent être rapidement impressionnantes. Dans [23] les auteurs donnent l'exemple d'une personne s'étant faite prendre la main dans le sac en 2001, après avoir vendu un objet du monde virtuel contre monnaie sonnante et trébuchante, puis utilisé une faille du serveur pour le reprendre à son nouveau propriétaire.

2.1.4 Passage à l'échelle

Dans le domaine des applications distribuées sur Internet, le passage à l'échelle est défini comme la capacité pour l'application de pouvoir rester performante malgré une montée en charge des ressources utilisées et du nombre d'utilisateurs. Même si les performances optimales ne peuvent être maintenues, des politiques de fonctionnement en mode dégradé, mais permettant toujours des conditions de jeu acceptables, doivent être prévues pour assurer la satisfaction de l'utilisateur.

Internet est un exemple d'application distribuée possédant de bonnes propriétés de passage à l'échelle : avec le nombre exponentiellement croissant d'utilisateurs, le réseau reste utilisable et performant. Cependant, même cette application distribuée «ultime» n'est pas exempte de problèmes liés à une forte augmentation des utilisateurs : en effet, le nombre de bits identifiant

les adresses IP commence à se montrer tellement insuffisant par rapport à l'ampleur qu'a pris le réseau mondial qu'il existe des situations de pénurie dans certains pays pauvrement dotés en nombre d'adresses IP, notamment sur le continent africain, ce qui a provoqué entre autre le projet IPv6 du nouvel Internet où les adresses seront désormais codées sur 128 bits [25].

Bien sûr, le problème se présente différemment pour les jeux massivement multi-joueurs. Tout d'abord, aucun ne prétend atteindre l'échelle d'Internet, les ambitions sont moindres en terme d'utilisateurs. L'intérêt d'atteindre un nombre de joueurs simultanés relevant du million est discutable en terme de *game-play*.

Selon les jeux, l'objectif affiché se compte en milliers de connexions simultanées (pour des jeux comme Everquest), ou en centaines (pour des jeux comme Neocron). La taille du monde dans lequel le joueur évolue a également son importance : le monde de Neocron serait surpeuplé s'il devait contenir le même nombre de joueurs qu'Everquest.

De plus, quand on parle de passage à l'échelle pour des jeux massivement multi-joueurs, le passage à l'échelle en nombre de joueurs simultanément présents dans le monde virtuel n'est pas le seul aspect à considérer. En effet, les joueurs ont souvent tendance à regrouper leurs avatars en masse dans certaines régions du monde, parce qu'elles présentent des facilités pour le commerce, parce que ce sont celles où ont lieu les combats entre les joueurs, ou que ce sont des zones hostiles où il est plus facile de trouver des personnages non-joueurs à combattre par exemple. Il faut que ces régions puissent également passer à l'échelle.

Les problèmes concrets posés par le passage à l'échelle ne concernent pas seulement la conservation des performances techniques dans le cadre des jeux multi-joueurs, mais sont aussi liés au fait que les concepteurs aiment à rendre ces jeux les plus réalistes possible. Par exemple, les avatars des joueurs ont un corps qui se comporte dans l'espace virtuel comme s'il était solide dans des jeux comme Neocron, ce qui n'est pas le cas dans Everquest. Les avatars des joueurs peuvent donc se cogner les uns contre les autres. En cas de surpeuplement d'une région du monde virtuel, il devient très vite difficile de se déplacer, et on peut ressentir très vite le même sentiment d'agoraphobie. Ce sentiment peut être également lié au «bruit» ambiant, les joueurs communiquant souvent entre eux par le moyen d'interfaces textuelles qui peuvent rendre difficile le suivi des conversations lorsque tout ce petit monde parle

en même temps.

2.1.5 Ressources d'une application distribuée sur Internet

Toutes les propriétés que nous avons précédemment décrites sont difficiles à réaliser simultanément pour une raison très simple : comme toute application informatique, un jeu massivement multi-joueurs consomme des ressources physiques qui sont limitées, et qui ont un coût : les ressources d'Internet et des machines hôtes de l'application distribuée.

Première difficulté, un jeu massivement multi-joueurs se joue sur Internet, qui est par essence un système distribué asynchrone [26] : on ne peut ni borner le temps nécessaire à la transmission des messages, ni s'appuyer sur une horloge partagée par tous les hôtes. Ce sont les protocoles classiques développés pour Internet qui permettent d'établir une bonne organisation des communications et d'y maintenir un semblant d'ordre.

Dans [26], les caractéristiques de performances d'un réseau sont définies comme suit :

- la *latence* (L) est le délai entre l'envoi d'un message par un processus et le début de sa réception par un autre processus. Elle peut être mesurée en comptant le temps écoulé entre l'envoi et la réception d'un message vide ;
- le *taux de transfert* (T) des données est la vitesse à laquelle les données peuvent être transférées d'un ordinateur à l'autre du réseau, et se décompte habituellement en bits par secondes ;

Dans le cas d'Internet, la latence est souvent le facteur clé des performances, car sa valeur va dépendre du chemin emprunté par le message, de routeur en routeur, et des problèmes de congestion éventuellement rencontrés en route. Le taux de transfert point à point des données dans un réseau est par contre une caractéristique physique de ce dernier.

Le *temps de transfert* tt d'un message contenant un nombre n de bits entre deux ordinateurs est alors défini par : $tt = L + n/T$, s'il ne s'agit pas de message trop longs qui seront alors souvent fragmentés selon les technologies utilisées.

La bande passante système d'un réseau est le volume total d'information qui peut passer par ce réseau en un temps donné. Cette caractéristique diffère du taux de transfert des données dans la mesure où elle est sujette à des variations selon la congestion du réseau et l'utilisation simultanée des différents canaux de communication pour un même message. Pour simplifier, la bande passante concerne principalement les réseaux hétérogènes comme Internet alors que le taux de transfert représente une mesure point à point entre deux composants physiques du réseau. Les deux notions peuvent correspondre sur réseau local.

Enfin, il ne faut pas oublier que les jeux massivement multi-joueurs sont également des programmes complexes sur chaque hôte de l'application distribuée, et que les programmes utilisent du *temps processeur* et de la *mémoire* pour effectuer tous leurs calculs.

La consommation de ces ressources augmente en fonction du nombre de participants à l'application, et des ressources insuffisantes peuvent poser problème pour le passage à l'échelle de l'application et pour la satisfaction de ses contraintes de synchronisation. En effet, plus de machines signifie plus de communications pour garantir la cohérence du monde virtuel, et donc un risque de congestion du réseau ne disposant pas de suffisamment de bande passante. De même, plus de joueurs signifie plus de calculs et de temps processeur consommé.

Dans [93], les auteurs décrivent, en se plaçant à un niveau un peu plus abstrait, l'ensemble des ressources consommées par une application de type monde virtuel par l'équation $Ressources = I \times D \times B \times V \times P$ où :

- I est le nombre de paquets d'informations transmis au travers du réseau de l'application ;
- D est le nombre moyen de destinataires pour chaque paquet d'information ;
- B est la quantité moyenne de bande passante nécessaire pour acheminer un paquet à destination ;
- V est la vitesse (inversement proportionnelle à la latence acceptable pour l'application) requise pour la délivrance d'un paquet d'information à destination ;
- P est le nombre moyen de cycles processeur requis pour recevoir et traiter un paquet d'informations.

Cette équation définit l'utilisation des ressources d'un monde virtuel : toute

technique visant à économiser la consommation d'une des ressource augmentera la consommation d'une autre de ces ressources.

2.1.6 Qu'est-ce que la jouabilité ?

Pour conclure sur cette description des caractéristiques techniques d'un jeu massivement multi-joueurs, nous définissons la *jouabilité* comme la qualité de l'immersion du joueur dans le jeu et son plaisir à évoluer selon le *game-play* élaboré par les concepteurs. La jouabilité s'obtient par une solution à un degré adéquat des contraintes techniques décrites tout au cours de cette section.

Atteindre la jouabilité se fait différemment selon les types de jeux, et il est nécessaire de faire correspondre les propriétés techniques des jeux avec le *game-play* désiré. Ainsi, les études présentées dans [9] et [85] permettent d'évaluer l'effet de mauvaises conditions de réseau sur les performances des joueurs dans *Unreal Tournament* 2003. De même, plusieurs études ont mesuré la qualité de la synchronisation nécessaire selon des *game-play* typiques :

- Dans [80], les auteurs étudient un jeu de course de voitures. Ils concluent que si le temps total de non-synchronisation observable par les «coureurs» dépasse 100ms les joueurs commencent à déceler le décalage, sans en être encore trop gênés, tandis que 50ms de décalage leur procurent une sensation d'instantanéité.
- Dans [91], les auteurs évaluent la latence acceptable à 500ms, dans un jeu de stratégie en temps-réel comme Warcraft III.
- Enfin, dans un jeu du style First Person Shooter, qui se joue beaucoup sur la rapidité de réaction des participants, les joueurs sont excédés dès que la latence avoisine les 150ms [3].

2.1.7 Synthèse

Après avoir défini les caractéristiques d'un jeu massivement multi-joueurs, nous allons étudier dans les sections qui suivent les différentes techniques actuellement utilisées pour tenter de donner des solutions aux problèmes techniques que nous venons de décrire. Nous allons voir qu'aucune solution n'est idéale, et que tenter de diminuer l'utilisation d'une des composantes

de l'équation des ressources ou de répondre à une contrainte forte sur les caractéristiques évoquées est toujours une affaire de compromis qui aura un impact négatif sur une autre caractéristique.

Certaines solutions aux problèmes que nous venons de décrire ne rentrent pas dans le cadre de cette thèse, et nous ne les aborderons pas dans ce qui suit. Elles concernent principalement la sécurité de l'application, et les problèmes de triche.

Tout d'abord, les techniques classiques de protection contre les attaques réseau auxquelles sont sujets les jeux comme toutes les autres applications Internet revêtent un champ d'étude très vaste, et déjà largement traité. Ces techniques (cryptage, utilisation de pare-feux, protection contre les attaques classiques, détection de l'intégrité du logiciel client, utilisation de traces pour détecter les comportements aberrants) sont des domaines que nous avons choisi de ne pas traiter pour mieux mettre en valeur les problématiques propres aux jeux massivement multi-joueurs.

De plus, bon nombre de solutions à des problèmes techniques se règlent par des choix de *game-design* : les jeux dont le *game-play* favorise le rassemblement d'un grand nombre d'avatars choisissent parfois de ne pas leur donner de modèle de collision (les avatars ne peuvent pas se pousser), d'autres comme City of Heroes [24] dupliquent les zones lorsqu'elles deviennent trop peuplées. De même, on peut poser une limite du nombre de joueurs pouvant se connecter simultanément, ce qui permet d'éviter les problèmes de passage à l'échelle non prévus. Des animations graphiques sur le logiciel client occupent le joueur pour qu'il ne remarque pas la latence. Enfin, pour empêcher un joueur qui réalise qu'il est en train de perdre l'avantage de s'échapper en se déconnectant brutalement, la plupart des jeux massivement multi-joueurs implémentent une solution de *game-design* qui consiste à laisser présent dans le monde virtuel l'avatar d'un joueur quelque temps après sa déconnexion, inactif puisqu'il n'est plus contrôlé par le joueur, mais toujours vulnérable. Et tant pis pour le joueur qui perd sa connexion au cours d'un combat à cause d'un problème matériel.

Certaines solutions dépendent également des choix des sociétés exploitant le jeu, notamment en ce qui concerne le *grief-play* : un comportement peut être possible à adopter par le joueur sans pour autant être considéré comme autorisé, tout simplement parce qu'il est techniquement impossible à interdire sans nuire à l'intérêt du jeu. Des politiques de traçage des actions des joueurs

et des objets du jeu peuvent permettre de mettre en oeuvre des mesures coercitives (bannissement du joueur) afin de faire respecter les règles.

2.2 Architectures de distribution

Dans cette section, nous allons passer en revue les différents types de solutions proposées en terme d'architecture de distribution des jeux massivement multi-joueurs. Comme il est impossible d'être exhaustif en la matière, tellement ce domaine de recherche est actif et les solutions différentes selon les priorités des concepteurs, nous allons en donner une typologie, et fournir à chaque fois quelques exemples représentatifs des propositions d'architectures dans le milieu académique.

Les sociétés exploitant des jeux persistants massivement multi-joueurs ont tendance à préférer une architecture de type clients-serveur au sens large : les joueurs sont les clients du jeu, et un arbitre, contrôlé par la société exploitant le jeu, gère la persistance de l'application et garantit les changements d'états ayant lieu dans le monde virtuel.

À cause des effets que les tricheurs peuvent avoir sur le monde virtuel, il est nécessaire de ne faire confiance aux clients que dans la mesure où cela n'est pas trop risqué pour la pérennité du monde virtuel. Il est plus raisonnable de faire vérifier par un serveur contrôlé par les exploitants du jeu leurs actions les plus critiques ayant un impact sur l'état du jeu. Certains travaux académiques proposent tout de même des modèles sans arbitre central, car ce dernier peut être particulièrement coûteux à mettre en oeuvre pour les sociétés exploitant le jeu.

2.2.1 Architectures logiques clients-serveur

Les architectures clients-serveur ont actuellement la préférence des sociétés exploitant des jeux massivement multi-utilisateurs, malgré le coût additionnel éventuel en temps de transmission des messages d'un joueur à un autre : ceux-ci doivent transiter par le serveur logique avant d'être finalement acheminés vers la machine du joueur destinataire. Ces solutions ont également un coût financier non négligeable, en terme d'hébergement et de bande

passante pour l'exploitant du jeu.

En effet, ces solutions permettent de gérer plus facilement la synchronisation de l'application, et de mettre en œuvre des politiques de vérification de la légitimité des actions des joueurs par un arbitre en qui on peut faire confiance. En bref, elles permettent de rendre le jeu exploitable.

Cependant, une architecture logique clients-serveur simple, où une seule machine serveur est l'interlocuteur de tous les clients, devient vite un goulet d'étranglement et peut nuire au passage à l'échelle du jeu. Le problème consiste donc à organiser la partie serveur du jeu, sous contrôle et à qui on peut faire confiance, en un ensemble de machines physiques capables de supporter un grand nombre de clients.

On peut classer ces différentes familles d'architectures par rapport à la façon dont elles permettent d'interagir avec les clients, c'est-à-dire par rapport à la couche visible vis-à-vis des connexions des participants au jeu.

2.2.1.1 Architectures à base de services

Les jeux massivement multi-joueurs ont un grand nombre de fonctionnalités en commun, même si ils ne sont pas traités de la même manière selon les jeux. Il est possible d'extraire certaines de ces fonctionnalités pour les traiter de manière indépendante du reste du comportement de l'application : par exemple, le service qui permet aux joueurs de dialoguer entre eux, ou le service permettant de répercuter les déplacement de l'avatar d'un joueur aux joueurs possédant les avatars à proximité. Outre la possibilité de régler finement par service la qualité de service désirée, cette division permet de réutiliser facilement des services plus ou moins génériques. (figure 2.1). De plus, si les services sont suffisamment indépendants les uns des autres, un niveau minimal de synchronisation et de communications entre les serveurs logiques responsables sera nécessaire. Chaque service gère une portion de l'état du jeu, et seules les portions redondantes entre les services devront être synchronisées.

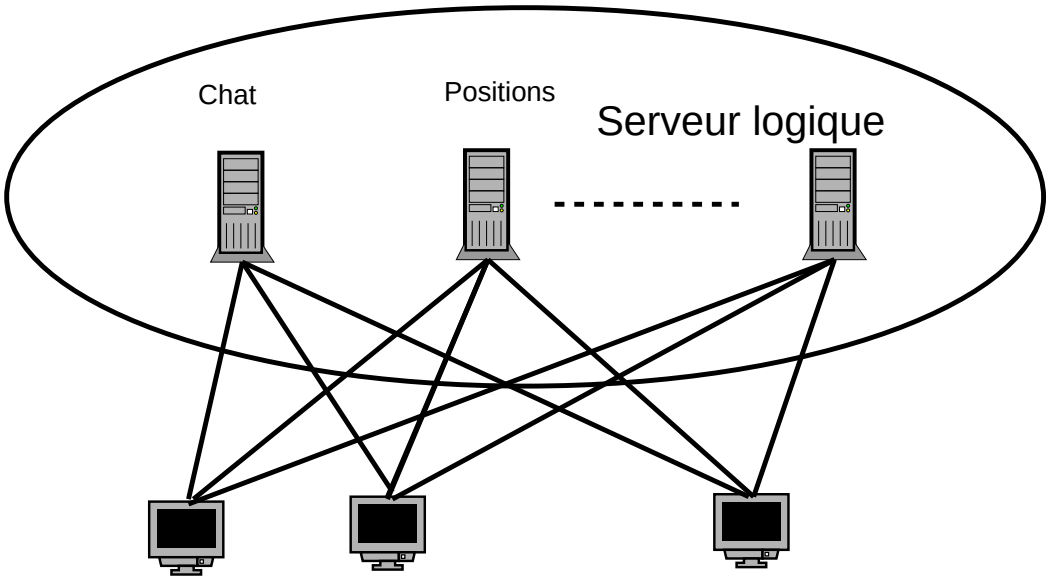


FIG. 2.1 – Serveur découpé en services

2.2.1.2 Architectures de type *proxies*

L'intérêt de cette solution est de réduire le temps de transmission des messages : elle se base sur l'attribution d'un interlocuteur pour chaque client (le *proxy*), par exemple selon la topologie physique du réseau. Son efficacité vis à vis de la rapidité de transmission des messages repose sur deux facteurs clés :

- le temps de transmission d'un message d'un client à son proxy est faible si ils sont proches physiquement ;
- le réseau entre les proxies dispose de très bonnes propriétés, en terme de latence et de bande passante. Cette propriété peut avoir un coût financier non négligeable pour le déploiement de l'application.

Les architectures de cette famille diffèrent principalement par les responsabilités attribuées à cette couche de *proxies* : les proxies peuvent être des serveurs de jeux complètement répliqués, contenant chacun la totalité de l'état du jeu, et synchronisés (figure 2.2). Cette solution peut permettre d'obtenir une meilleure rapidité de réponse à des requêtes des clients, mais peut coûter cher en bande passante à cause des communications entre les serveurs. Elle

peut également compliquer la synchronisation des différents serveurs logiques.

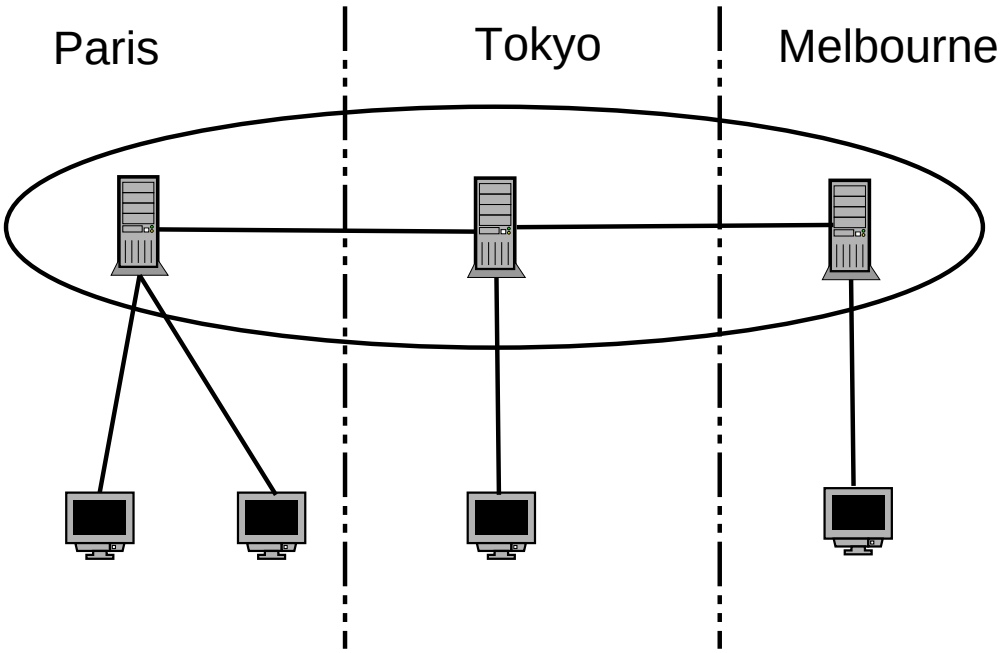


FIG. 2.2 – Serveur découpé en proxies répliqués

A l'autre extrême des possibilités, les proxies peuvent être juste une couche intermédiaire se chargeant d'acheminer les messages entre les clients à bon port, et les messages modifiant l'état du jeu vers une autre couche de serveur logique chargée de garantir la cohérence de l'application.

GISA [1] (Generic Internet Scalable Architecture) est une architecture à deux couches logiques, dont la première couche, dénommée *Concentration Tier* est en charge des connexions des clients, et des communications avec le reste de l'application. Une deuxième couche, dénommée *Server Tier* consiste en un ensemble de serveurs logiques synchronisés en charge de la maintenance de l'état du jeu. Les *Concentrators* ne font pas simplement du routage, ils permettent de filtrer certains événements venant des clients et sont munis d'un service d'adaptation dynamique à la bande passante de chacun d'entre eux.

Dans [68], les auteurs proposent une architecture de serveur logique hiérarchisée, où la gestion de certains éléments de l'état du jeu est déportée sur les proxies tandis que la totalité de l'état du jeu est maintenue sur la couche supérieure, de manière centralisée.

L'architecture en miroir proposée dans [27] se base sur une seule couche de proxies, auxquels les clients se connectent selon la topologie du réseau. Les proxies contiennent chacun une copie totale de l'état du jeu et sont synchronisés. La solution s'inspire, pour l'architecture serveur, d'une construction similaire à celle décrite dans la figure 2.2.

2.2.1.3 Architectures basées sur un découpage de zones virtuelles

Cette solution, qui a pour but de faciliter la synchronisation de l'état du jeu et d'économiser la bande passante entre les différents serveurs logiques est la plus en vogue dans l'industrie. Elle est basée sur une constatation simple : dans les mondes virtuels comme dans le monde réel, on communique et on interagit avec les gens qu'on perçoit. Elle consiste à diviser le monde virtuel en régions, chaque région étant gérée par un serveur logique différent. Chaque région dispose donc d'une portion indépendante de l'état du jeu à traiter, incluant tous les objets virtuels s'y trouvant (Figure 2.3. Chaque serveur de zone peut donc disposer d'une liste communes de services liés à la localisation dans le monde virtuel. Il peut gérer les déplacements et transactions s'y déroulant. Tout comme l'architecture basée sur un découpage en proxies, il peut être complètement découplé des autres serveurs et gérer lui même la persistance avec un support externe de base de données, ou dépendre d'un serveur central qui se charge de synchroniser certains états du jeu et ses aspects persistants. Cela implique qu'au cours de leurs déplacements dans le jeu les joueurs se déconnectent et se reconnectent de manière plus ou moins transparente à un autre serveur, lorsqu'ils quittent une région pour entrer dans une autre. Cette architecture est tellement populaire qu'elle a donné naissance à une tactique particulière de fuite, pour les joueurs dont l'avatar est en danger : le *zoning* consiste à quitter précipitamment une région du monde virtuel où son avatar est poursuivi par un *personnage non-joueur* menaçant. Ce *personnage non-joueur* n'étant en général pas conçu pour changer lui aussi de serveur, faisant partie de l'état du jeu géré par le serveur de la zone, l'avatar est sauf.

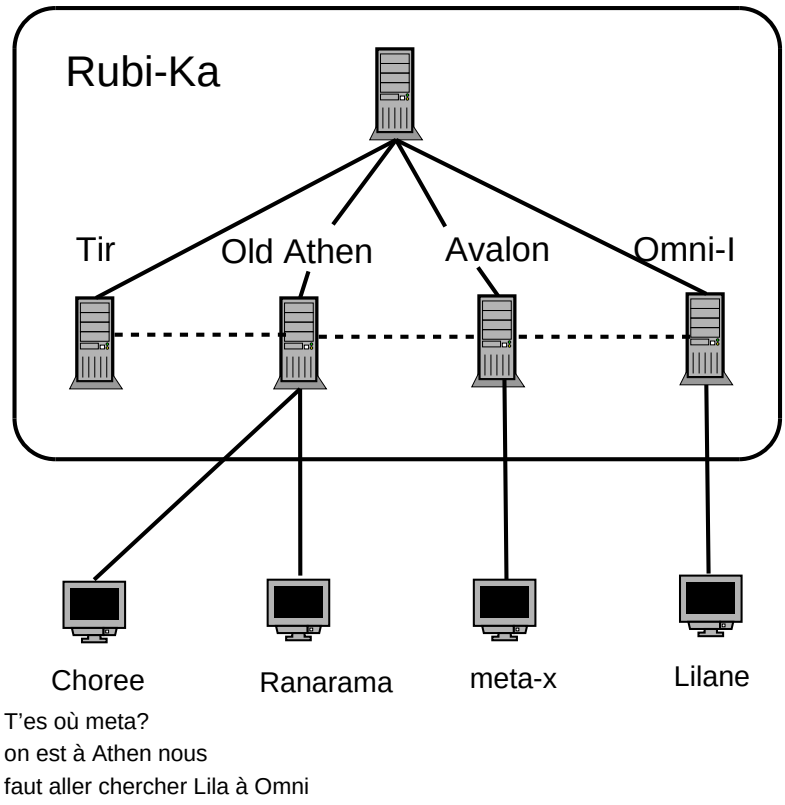


FIG. 2.3 – Architecture basée sur les zones géographiques du monde virtuel

Un autre avantage collatéral est qu'en cas de défaillance d'un serveur logique, seule une zone du monde est inaccessible aux joueurs.

Le système RING a été adapté pour une telle utilisation dans un souci de passage à l'échelle sur Internet, avec pour objectif de réduire les coûts en bande passante des communications entre les serveurs de zone auxquels les clients sont connectés [45].

2.2.2 Architectures *peer to peer*

Ce type d'architecture a pour but de réduire le coût de déploiement et d'exploitation des jeux massivement multi-joueurs pour les sociétés qui ne peuvent investir de grosses sommes dans des architectures clients-serveur.

Ces architectures sont composées uniquement des machines des joueurs, les pairs, qui représentent le monde virtuel. Tous ces pairs se communiquent entre eux les modifications de l'état du monde. (Figure 2.4). La plupart des

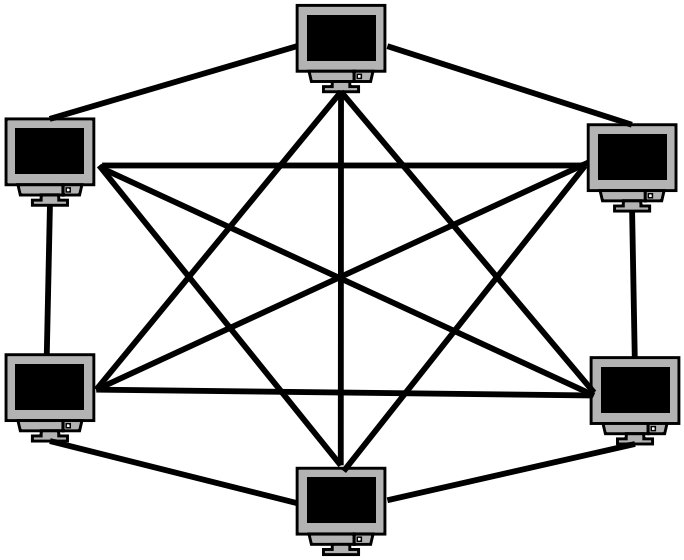


FIG. 2.4 – Architecture *peer to peer*

travaux en ce sens proviennent de la recherche induite par les mondes virtuels non commerciaux, parfois sur réseau propriétaire où les problèmes relatifs à la nature d'Internet n'interviennent pas, et des recherches dans le domaine du calcul distribué.

Une architecture purement *peer to peer* peut difficilement passer à l'échelle : si tous les clients doivent communiquer entre eux et diffuser leurs changements d'états, la consommation de bande passante de l'ensemble du réseau grandit de manière spectaculaire dès qu'un joueur se connecte à un jeu déjà peuplé. De même, si aucune précaution n'est prise à cet effet, il est particulièrement délicat de s'assurer que des incohérences ne vont pas s'introduire dans l'état du jeu, puisque rien ne garantit a priori que tous les pairs vont recevoir les mises à jours des différents clients dans le même ordre, et à partir du même état de référence.

MiMaze, développé à l'INRIA, est un exemple de jeu multi-joueurs sur

Internet qui repose sur une architecture de cette famille [31], mais qui utilise des techniques permettant de résoudre les problèmes de passage à l'échelle. Nous reviendrons plus tard sur cette application qui possède des propriétés intéressantes.

La plupart des propositions d'architectures totalement *peer to peer* pour des jeux massivement multi-joueurs ne donnent aucune solution aux problèmes de sécurité et de triche qu'elles posent par nature, laissant cet aspect pourtant crucial à des projets de recherche ultérieurs.

Dans [52], les auteurs proposent une architecture purement *peer to peer* dans l'objectif de construire un jeu massivement multi-joueurs passant à l'échelle de manière importante tout en gardant un bon niveau de réactivité, en utilisant pour résoudre ce problème une construction mathématique élaborée. Les résultats annoncés concernant le défi relevé sont très satisfaisants. Malheureusement, les auteurs préviennent dès le début de l'article : ils font confiance aux clients et les échanges de messages sont suffisants pour assurer la maintenance de l'état de l'application. En d'autres termes, les aspects de sécurité et de persistance ne sont absolument pas pris en compte.

Dans [55], l'architecture *peer to peer* proposée a également pour but de résoudre le problème du passage à l'échelle de l'application. A cet effet, chaque participant au monde a la responsabilité de la gestion de l'état du jeu concernant une zone géographique du monde virtuel. L'inconvénient majeur de cette solution est que si l'un des hôtes de l'application est sujet à une défaillance technique, c'est toute une zone du jeu qui disparaît. De plus cette solution, là encore, est plus adaptée au cas de joueurs se faisant confiance car chacun pourrait si l'envie lui prenait tricher sur la gestion de l'état du jeu dont il est responsable. Les auteurs pensent améliorer cet aspect dans de futurs travaux en utilisant des mécanismes de réputation mais il nous semble probable que même totalement aboutie, l'idée n'est pas assez sûre pour convenir à un jeu massivement multi-joueurs persistant car il resterait sensible à des attaques d'un grand nombre de joueurs coordonnés. Toutefois, la solution nous semble intéressante pour les jeux par session, par exemple, un jeu de stratégie temps réel massif en nombre de joueur mais se jouant par parties.

De plus, beaucoup de ces architectures inspirées par le domaine de la simulation ont été inventées pour être déployées en réseau local. Dans ce cas, on peut s'appuyer sur des protocoles de communications de groupe plus robustes que ceux proposés sur Internet et que nous étudierons plus loin, ce

qui n'est malheureusement pas le cas des jeux massivement multi-joueurs.

Les problèmes de cohérence de l'état du jeu ayant un impact moindre pour les jeux multi-joueurs non persistants, ces solutions peuvent être plus intéressantes dans le cas des jeux éphémères, se jouant par session sur Internet.

2.2.3 Synthèse

Dans la pratique et selon les performances recherchées, il est souvent adéquat de mélanger allègrement les différentes architectures que nous avons présentées, pour organiser l'application de manière à mieux répondre aux ressources jugées critiques : les architectures centralisées souffrent de problèmes de congestion, les solutions non centralisées posent des problèmes de consistance, de cohérence de l'état du jeu, et les solutions basées sur des *proxies* peuvent améliorer la latence mais sont plus coûteuses.

Par exemple, un premier découpage en services génériques (service de *chat* ou de communication audio) peut être suivi d'un découpage en *proxies* chargés du routage de l'information entre des serveurs de zone se partageant l'état du jeu.

Il est également possible de superposer deux architectures différentes, gérant chacune de manière différente l'état du jeu. Par exemple, on trouve dans [82] une architecture hybride, *peer to peer* avec serveur. Le rôle du serveur y est limité à un arbitrage du jeu. Chaque interaction d'un joueur avec le monde virtuel est envoyée à tous les autres clients, mais également au serveur qui joue un rôle d'arbitre : il calcule et simule l'état du jeu, et s'il détecte un problème de synchronisation, il intervient auprès des clients. Ce modèle garde les propriétés de rapidité de communication des architectures *peer to peer*, conserve les garanties de synchronisation des architectures clients-serveur, mais garde des problèmes de passage à l'échelle, même quand tout se passe bien et que le serveur n'a pas à intervenir pour re-synchroniser l'application.

En conclusion, il n'y a pas d'architecture idéale, toutes ont leurs avantages et inconvénients, améliorant une caractéristique jugée critique pour dégrader le traitement d'une autre, augmentant le coût financier du jeu en améliorant le passage à l'échelle, améliorant le temps de transmission des messages entre

joueurs en rendant la synchronisation plus problématique, et ainsi de suite.

Il ne peut donc pas exister d'architecture générique pour les jeux massivement multi-joueurs

2.3 Modèles de communication

2.3.1 Politiques globales de mise à jour de l'état du jeu

Différentes politiques de communication entre les hôtes de l'architecture distribuée d'un jeu massivement multi-joueurs peuvent être utilisées, selon les propriétés jugées les plus critiques. De ces politiques de mises à jour dépend la synchronisation de l'application, et le degré de cohérence global de l'application est fortement lié à ce choix. Voici une classification non exhaustive de ces différentes politiques de mise à jour.

2.3.1.1 Régénération fréquente de l'état du jeu

Le principe général de ce modèle bien décrit dans [93] est simple : chaque hôte de l'architecture de distribution responsable de la gestion d'une partie de l'état du jeu envoie régulièrement à tous les autres hôtes les valeurs à jour de ces états, de la manière la plus rapide (et donc non fiable) possible sans se poser la question de savoir si la valeur d'un état particulier a été modifié depuis la dernière communication, ou si un des destinataire a reçu la valeur précédemment communiquée (les communications ne sont pas forcément ordonnées, et il en est donc de même pour les mises à jour).

Ce choix sacrifie la synchronisation de l'application au profit de la rapidité de mise à jour, surtout dans des mauvaises conditions de réseau : la cohérence des états du jeu vus de chaque hôte de la distribution souffre avec la perte et le non ordonnancement des messages.

Ce modèle a beaucoup été utilisé à l'époque des premiers FPS , qui étaient principalement destinés à être joués sur réseau local en mode multi-joueurs : il est en effet assez simple de rajouter ce modèle à un jeu mono-joueur pour le transformer en un jeu multi-joueurs.

La consommation de bande passante que ce modèle implique a vite donné lieu à l'application des techniques de communications de groupe, que nous étudierons plus loin. Par exemple, l'adaptation de RING [45] qui utilise originellement ce modèle de communication avec des techniques d'optimisation adéquates le rend viable pour une application plus massive sur Internet.

2.3.1.2 Temporisation

Les messages transmis entre les hôtes d'un jeu massivement multi-joueurs peuvent parfois être très nombreux et de petite taille. Lorsque le besoin se fait sentir d'économiser la bande passante et que la rapidité de mise à jour n'est pas critique, il est alors possible de travailler sur la couche réseau de l'application pour rassembler les informations en un seul message [93]. Cette technique permet d'économiser sur la taille des en-têtes des messages qui peuvent représenter la plus grande partie du message transmis.

- On peut décider d'une taille optimale de quantité d'information à envoyer au destinataire, et attendre que ce quorum soit atteint avant de construire le message et de l'envoyer. Cependant, si le rythme des mises à jour est irrégulier, on risque d'introduire un délai de mise à jour gênant.
- On peut également poser un intervalle fixe auxquels les messages seront assemblés et envoyés. On maîtrise ainsi bien mieux le sacrifice de latence qui sera fait, mais si peu de messages sont produits pendant l'intervalle il y a peu de chance de faire des économies.

Il peut donc être intéressant, pour les applications se mettant à jour de manière très irrégulière, d'utiliser les deux bornes simultanément, et de concaténer les messages selon la première borne (temporelle ou quorum d'informations) atteinte. Ainsi, en fixant ces bornes de manière adéquate, on peut équilibrer selon les besoins spécifiques de l'application le compromis à trouver entre la bande passante et la rapidité des mises à jour.

2.3.2 Politiques de mise à jour état par état

Les deux premiers modèles que nous allons aborder rentrent dans la catégorie des modèles à objets distribués. Dans ces modèles, l'application est vue comme un ensemble d'objets indépendants qui peuvent être invoqués sur

chaque hôte par d'autres objets distants. C'est l'extension du paradigme de programmation objet aux applications distribuées.

2.3.2.1 Invocation de méthode distante

Dans cette approche, les objets peuvent invoquer des méthodes d'objets distants tout comme ils peuvent invoquer des méthodes d'objets locaux dans le paradigme de programmation objet classique. Les communications sont synchrones, c'est-à-dire que l'objet qui invoque attend la réponse de l'objet distant. Ce modèle a donné lieu à de nombreuses implémentations dans le domaine des applications distribuées généralistes. Par exemple, Java RMI [88] en donne une implémentation : les invocations se font selon la même syntaxe pour les objets locaux et distants. Le programmeur doit utiliser une interface particulière pour représenter les objets distants, ce qui lui permet de gérer les problèmes particuliers à ce type d'invocation au travers d'exceptions spéciales. Nous étudierons plus en détail plus tard un *framework* utilisant un modèle apparenté (voir 3.3, page 90).

2.3.2.2 Modèle diffusion - souscription

Ce modèle est guidé par les événements se produisant pour les objets. Un objet diffuseur permet à des objets distants souscripteurs de s'abonner, pour recevoir certains événements. Lorsque l'événement pour lequel un souscripteur s'est abonné se produit, il est notifié par le diffuseur.

Les notifications envoyées par les objets diffuseurs, producteurs d'événements, sont envoyées de manière asynchrone aux souscripteurs.

Ce modèle de communication comporte plusieurs implémentations classiques, non dédiées aux jeux massivement multi-joueurs. Par exemple, *Jini* [4] permet à un objet d'une machine virtuelle Java de souscrire à un objet résidant sur une autre machine virtuelle Java et d'en recevoir les événements.

En ce qui concerne les jeux massivement multi-joueurs, le système Mercury [12], basé sur ce modèle, a été construit pour répondre à des impératifs de passage à l'échelle. De même, les auteurs de [40] proposent d'utiliser ce modèle conjointement avec une division en zones du monde virtuel, et proposent une catégorisation statique des objets diffuseurs selon que les événe-

ments qu'ils produisent modifient le monde virtuel ou non.

2.3.2.3 Réplication d'états

Le principal inconvénient du modèle diffuseur-souscripteur est que rien n'empêche un client de jeu modifié de souscrire aux événements d'un objet, et qu'ils peuvent donc constituer un risque de triche en permettant à un joueur d'obtenir des informations lui donnant un avantage injuste.

Pour répondre à ce problème de sécurité, l'auteur de [48] propose un *framework* basé sur un modèle plus fin de réplication d'états, incluant également une notion de priorité, construit sur une architecture clients-serveur utilisant des proxies. Le modèle de communication est la encore guidé par les événements, et l'assignation des objets devant recevoir des notifications est faite côté serveur, de la manière décidée par le développeur.

2.3.3 Techniques de communications de groupe

Ces techniques ont pour but d'optimiser la consommation de bande passante, en réduisant le nombre de destinataires à qui les mises à jour d'états doivent être envoyées.

2.3.3.1 Le protocole multicast sur Internet

La première chose qui vient à l'esprit, quand on pense à réduire la bande passante utilisée pour envoyer le même message à un grand nombre de personnes est le protocole multicast, dont il existe une implémentation IP. Les récepteurs doivent être abonnés à un *groupe de multicast*, identifié par une adresse IP. L'expéditeur du message envoie la mise à jour à cette adresse IP, et c'est le protocole qui se charge du routage de l'information aux abonnés. L'utilisation de ce protocole a donné lieu à de nombreux travaux dans le domaine des mondes virtuels, surtout ceux étant construits pour fonctionner sur des réseaux propriétaires. Par exemple, [65] propose tout simplement de partitionner le monde en zones géographiques, et d'attribuer à chaque zone une adresse de multicast différente pour le routage des mises à jour de la portion de l'état du jeu concernant cette zone.

Malheureusement, l'implémentation d'IP-multicast pose quelques problèmes pour son utilisation dans le cadre d'un jeu massivement multi-joueurs. Tout d'abord, l'envoi d'un message à une adresse multicast n'est pas sécurisé, un intrus peut donc perturber le déroulement de l'application en envoyant des messages à cette adresse. De même, un joueur disposant d'un client modifié peut joindre un groupe de multicast et obtenir des informations qui peuvent lui permettre d'obtenir un avantage injuste. Les adresses multicast sont également peu nombreuses sur l'Internet actuel (IPv4), et les sessions doivent être annoncées (sur ce point précis, IPv6 fournira des améliorations [25]).

De plus IP-multicast repose sur des transmissions non fiables et non ordonnées. En d'autres termes, il n'y a aucune garantie que le message soit bien acheminé à tous les destinataires abonnés, ce qui peut être nécessaire pour certains messages critiques.

Enfin, les performances actuelles de l'implémentation sont insuffisantes : joindre un groupe de multicast sur Internet est une opération encore trop longue pour permettre de baser des modèles de communications nécessitant des changements fréquents d'abonnement à une adresse multicast. Les auteurs de [64] fournissent une étude très pertinente des limitations actuelles de l'implémentation du multicast sur Internet ainsi que des pistes pour son amélioration.

Toutefois, certains jeux ont relevé le défi. Mimaze [30], dont nous avons déjà parlé en évoquant les architectures *peer to peer* est un jeu entièrement basé sur le multicast, afin d'améliorer le passage à l'échelle du jeu. Les auteurs utilisent des techniques d'extrapolation pour remplacer les messages éventuellement perdus et une horloge partagée pour synchroniser l'application et améliorer la consistance. Cette solution fournit des qualités de synchronisation suffisantes pour le jeu Mimaze, mais pas pour un jeu de style FPS où les interactions entre les joueurs sont plus nombreuses [27].

2.3.3.2 Le multicast au niveau applicatif

Il existe de nombreuses propositions de multicast au niveau applicatif pour les applications distribuées, proposant des propriétés qui manquent à IP-multicast. Dans ces techniques, le routage des informations est calculé sur les hôtes de l'application, et des propriétés d'ordonnancement ou de réachemi-

nement des paquets perdus, qui manquent au protocole IP-Multicast, peuvent alors être utilisés. Les chercheurs du domaine font la distinction entre les solutions purement applicatives, où le routage est effectué sur chaque hôte de l'application, et les solutions reposant sur une architecture à base de *proxies* chargés de l'implémentation du protocole de distribution. On peut trouver dans [62] un comparatif de quelques unes de ces techniques. Certaines de ces solutions ont été utilisées dans le domaine des mondes virtuels. Par exemple, DIVE [44], dispose d'un module permettant d'utiliser un protocole multicast fiable, au niveau applicatif [2]. Yoann Fabre propose également dans [36] d'utiliser la bibliothèque *Extended Virtual Synchrony* [71] pour la conception d'une solution générique pour créer des applications de type «monde virtuel». Cependant, la plupart des solutions proposées dans le cadre de ces applications spécifiques sont généralement très spécialisées pour le domaine d'application. Ce sont les techniques de gestion des intérêts dont nous allons parler dans ce qui suit.

2.3.3.3 Les techniques de gestion des intérêts au niveau applicatif

Ces techniques consistent à regrouper les différents hôtes de l'application selon leur intérêt pour certaines mises à jour ou événements produits sur d'autres hôtes de l'application [70, 59, 37].

Le concept repose sur les notions de *focus* et de *nimbus* des objets du monde virtuel. Nous illustrerons ces concepts en considérant comme objet l'avatar d'un joueur :

Le focus d'une entité du monde virtuel représente tous les objets sur lesquelles l'avatar doit recevoir des informations. Transposé dans le cadre de la géographie d'un monde virtuel, ce sont les objets de son «champ de vision». Le nimbus représente toutes les objets à qui il doit envoyer les informations sur son changement d'état. Pour la même analogie spatiale, c'est la zone d'où il est «visible» par un autre objet. Dans l'idéal, on traite chaque information séparément, et on l'envoie uniquement aux entités dont le nimbus a une intersection avec le focus de l'émetteur. L'inconvénient de ce modèle est que des calculs potentiellement importants selon le nombre de destinataires sont faits à chaque fois que l'état du jeu est modifié. Sa complexité dépend de sa

[2] Dive : manuel utilisateur : `http ://www.sics.se/dive/manual/sid2.html`

précision : si on découpe le monde en un maillage de carrés, il sera moins complexe mais moins précis que si on utilise des figures polygonales se rapprochant plus de l'idéal du cercle. C'est acceptable seulement quand il n'est pas indispensable de supporter un passage à l'échelle en nombre de clients, et quand le monde n'est pas trop riche et dynamique.

Dans la pratique, et pour permettre un bon passage à l'échelle de ces techniques coûteuses en temps de calcul, ces services sont souvent installés côté serveur sur la couche de l'architecture en communication avec les clients. Par exemple, dans GISA [1] un filtrage des intérêts des clients, est utilisé sur les *proxies* d'une architecture serveur centralisée.

Ces services peuvent également être adaptés dynamiquement aux capacités de traitement de l'application et permettre un équilibrage dynamique de la charge. Par exemple, si un client dispose d'une connexion de moindre qualité, il est possible de réduire ses centres d'intérêts en définissant un mode dégradé basé sur des priorités qui lui permet de recevoir les communications les plus importantes vis-à-vis du *game-play*. Cette philosophie s'inspire des techniques utilisées pour le rendu graphique des jeux, qui doivent pouvoir s'adapter aux capacités des différents matériels utilisés par les joueurs, en diminuant la qualité du traitement de l'image pour gagner en rapidité. Cette possibilité est évoquée dans [48]. De même, dans le cadre d'une architecture de serveur logique basée sur les zones géographiques du monde virtuel, si le serveur souffre d'un soudain surpeuplement, une réadaptation dynamique du filtrage des intérêts des clients basée sur une telle définition des priorités peu permettre à l'architecture de continuer à fonctionner de manière satisfaisante en sacrifiant un peu de temps de calcul à une meilleure utilisation de la bande passante disponible.

2.4 Techniques de masquage de la latence

Les techniques que nous allons aborder dans cette section ont deux principaux contextes d'application.

Les jeux doivent présenter un état du jeu le plus cohérent possible à chaque joueur pour être *jouables*. Or, les temps de transmission des messages sur Internet peuvent souvent dépasser la limite acceptable pour le joueur.

La désynchronisation temporaire de sa vision de jeu par rapport à l'état courant de l'application peut l'empêcher d'adapter ses actions à celles de ses partenaires qu'il reçoit trop tardivement. Cette désynchronisation peut également le frustrer quand ses propres actions sont faussées car ce qu'il perçoit ne correspond pas à l'état jugé valide par l'arbitre de l'état global du jeu. Il devient donc nécessaire d'adopter des stratégies de masquage de ce problème de cohérence, dû à l'inévitable latence réseau.

De plus, les protocoles de communications rapides qu'on utilise dans le cadre de ce type de jeu peuvent perdre des informations sur Internet. Ces inconvénients sont d'importance imprévisible, et inhérents à la nature même d'Internet, on ne peut que faire avec. Les techniques de masquage de la latence ont pour but non pas de résoudre ces problèmes, mais de les masquer à l'utilisateur en essayant de lui fournir l'illusion que toutes les informations sont bien arrivées.

Dans [10], un article décrivant les solutions mises en oeuvre pour la réalisation du FPS à succès *Half-Life* [51], l'auteur décrit les trois principales techniques utilisées pour masquer la latence : l'introduction d'un délai artificiel, la compensation côté serveur, et le *Dead-reckoning* (voir glossaire). Nous allons passer en revue ces différentes méthodes et discuter d'autres travaux à leur sujet.

2.4.1 Introduction de délai

Ces techniques consistent à synchroniser «de force» l'application sur chaque client, en introduisant un délai suffisant pour que tous les messages nécessaires à produire l'état du jeu arrivent à destination avant de fournir une représentation de cet état au client. Le jeu tel qu'il est perçu du point de vue du joueur a donc toujours un temps de retard par rapport aux dernières modifications de l'état du jeu, qu'elles soient déjà disponibles sur la partie cliente de l'application ou non. Son principal avantage est que les joueurs ont plus de chances d'avoir la même vue sur l'application, pourvu que le délai introduit soit supérieur ou égal à la latence du système. Son principal inconvénient est qu'un client malintentionné peut modifier son programme afin d'exploiter les informations présentes mais qu'il n'est pas censé percevoir à son avantage dans le jeu. De plus, le choix de la durée du délai est crucial : plus il est grand, plus on s'approchera d'une vision idéale de l'appli-

cation, où tous les joueurs percevront le même monde, mais plus le décalage entre ses actions sur le monde et leurs conséquences sur l'état partagé par tous sera perceptible, et potentiellement gênant pour un *game-play* rapide et réactif. L'équilibrage de ce compromis est intrinsèquement lié aux choix de *game-design*, et donc propre à chaque jeu. Les auteurs de [80], déjà cité, fournissent une étude des délais produisant des *game-play* jouables pour des types de jeux classiques (courses de voiture, FPS).

2.4.2 Compensation côté serveur

Bien nommées, ces techniques visent à corriger les conséquences de la latence pour les joueurs dans le cadre d'une architecture logique clients-serveur.

La technique classique décrite dans [10] consiste à évaluer l'effet des messages transmis par chaque client dans le contexte de l'état du jeu dans lequel il s'est produit, en retrouvant cet état passé à l'aide d'une estimation de la latence de ce client. Coûteuse en calcul côté serveur, cette solution peut néanmoins être profitable et réduire le sentiment de frustration des joueurs.

Dans [49], les auteurs proposent un service d'échange de message équitable qui peut être placé sur la couche visible des clients dans le cadre d'une architecture logique clients-serveur. Le but est d'ordonner les messages provenant des différents clients afin que ce ne soit pas toujours celui qui a la plus faible latence qui emporte la mise, sans pour autant avoir besoin d'utiliser une horloge partagée. À cet effet, lorsqu'une mise à jour d'état est envoyée par le serveur, la réponse des clients est prise en compte avec un temps de réaction qui est utilisé pour coordonner les réponses des différents clients.

2.4.3 Dead-reckoning

Cette technique, particulièrement bien décrite dans [59] et [93], peut être utilisée dans le cadre d'applications construites sur une architecture *peer to peer* [31] comme clients-serveur. Elle a été très largement utilisée, dans le cadre des jeux de style FPS industriels comme dans l'académique, pour permettre un rendu fluide des déplacements des avatars des joueurs distants. Elle consiste à extrapoler la nouvelle valeur d'un état du jeu, dont la mise à jour tarde à arriver ou s'est perdue dans le réseau, à partir de ses valeurs

précédentes. Bien sûr, la valeur extrapolée peut se révéler finalement incorrecte auquel cas il faudra la corriger en évitant de faire sauter brutalement l'objet mal déplacé de sa position extrapolée à la nouvelle. Il faut donc mettre au point deux algorithmes complémentaires, l'un pour la prédiction, l'autre pour la correction éventuelle de cette prédiction.

Par exemple, dans [81], les auteurs comparent les performances de deux algorithmes de prédiction dans différents styles de jeu (FPS, courses de voitures). Le premier, très classique, est basé sur la dérivation de polynômes représentant les caractéristiques des déplacements des avatars des joueurs pour obtenir leurs vitesse, accélération, et courbe instantanées afin d'en déduire leur position suivante. Le deuxième peut paraître plus surprenant, et se base sur les états passés des périphériques de contrôle des joueurs (souris, clavier...) pour prédire ses prochaines actions (et en déduire par exemple également la prochaine position de son avatar, mais cela peut aussi être d'autres actions concernant les objets qu'il contrôle).

La méthode classique des polynômes dérivés étant basée sur des données instantanées et se révélant donc souvent insatisfaisante, l'auteur de [94] propose une méthode hybride, permettant un choix dynamique de l'ordre du polynôme, utilisant un polynôme du 1er ordre quand la vitesse est petite ou nulle, ou que l'accélération change fréquemment. Sinon, on s'autorise un polynôme d'ordre supérieur qui donnera des résultats plus précis (au prix de plus de consommation de temps de calcul). Ce protocole permet également de baser l'information non sur des données instantanées, mais sur des moyennes sur une période de temps, ce qui contribue à garantir des résultats plus stables.

Plus récemment,les auteurs de [2] proposent d'améliorer la pertinence des résultats obtenus par l'utilisation d'une horloge partagée.

Les algorithmes de correction ont eux à résoudre un compromis fluidité et naturel contre rapidité de la correction. À un extrême de l'étendue des possibilités, si la re-synchronisation de l'application doit être faite le plus vite possible, il est toujours possible de la corriger brutalement. À l'autre, calculer une courbe permettant de déplacer moins rapidement mais plus naturellement l'objet à sa nouvelle position permet de maintenir le joueur dans l'illusion d'une bonne synchronisation au prix justement de la synchronisation et d'un peu de temps de calcul.

Les principaux inconvénients du Dead-reckoning sont décrits dans [6]. Tout d'abord, par nature, ces techniques introduisent des inconsistances dans le jeu tel que perçu par les joueurs par rapport à l'état de référence de l'application, ce qui peut mener à une certaine frustration et à des décisions considérées comme injustes du *game-play* (le joueur croyait viser juste, mais en réalité il avait tiré à côté de la cible dont la position était mal extrapolée). De plus, un joueur malintentionné peut modifier son client afin d'exploiter l'algorithme à son avantage : il peut ralentir temporairement la diffusion de ses déplacements, simulant ainsi un problème réseau. Cela provoquera chez les autres joueurs l'utilisation de l'algorithme d'extrapolation, tandis que le tricheur prendra lui bien garde à ne pas faire correspondre ses déplacements à la prédiction résultant de l'algorithme. Ainsi, les autres joueurs auront une mauvaise représentation de sa position et bien des difficultés à le viser.

2.5 Conclusion

Grâce à ce catalogue des techniques utilisées pour la mise au point des jeux massivement multi-joueurs, qui met en balance les bénéfices apportés par chaque solution par rapport à ses inconvénients, nous pensons avoir démontré dans ce chapitre le pivot de notre réflexion : les jeux massivement multi-joueurs sont des applications complexes à mettre au point, et chaque choix technique visant à l'amélioration d'une caractéristique jugée critique pour la jouabilité se paye par une moins bonne résolution d'une autre caractéristique.

Réaliser un jeu massivement multi-joueurs est une affaire de compromis, dont la solution est intrinsèquement liée au type de jeu que l'on veut réaliser.

Chapitre 3

Modèles, méthodes et outils

3.1 Concurrence

Les applications d'aujourd'hui sont plus gourmandes et complexes que celles des débuts de l'informatique. C'est vrai dans le domaine des jeux-vidéo comme ailleurs : le temps où un jeu était programmé en une seule boucle d'exécution (lire les événements, calculer le nouvel état du jeu, afficher le nouvel état du jeu, répéter ces trois étapes...) est maintenant loin. Pour décrire un programme complexe, il est beaucoup plus simple d'utiliser une approche concurrente de la programmation, où chaque tâche est décrite séparément par le programmeur, et où la plate-forme d'exécution gère l'exécution de ces tâches en parallèle à l'aide d'un ordonnanceur (que ces tâches soient des processus ou des processus légers[1] selon les circonstances).

Le travail présenté dans cette thèse propose donc, pour chaque hôte de la distribution d'un jeu massivement multi-joueurs, un modèle d'exécution concurrent, divisant ainsi l'exécution du programme en plusieurs tâches sur la machine. Pour expliquer ultérieurement nos choix, nous allons passer en

[1] Nous définissons comme processus une tâche indépendante, par opposition aux processus légers aussi appelés *threads* qui partagent mémoire et ressources lorsqu'ils sont gérés par le même processus. Pour communiquer entre eux, les processus doivent mettre en œuvre des mécanismes externes comme des canaux de communication ou des mémoires partagées. La communication entre processus légers est facilitée par le partage d'un espace de données

revue différentes approches de la programmation concurrente.

3.1.1 Les modèles classiques

Les deux modèles de programmation concurrente classiques sont ceux qui ont été conçus pour créer les systèmes d'exploitation, et sont maintenant utilisés par de nombreux langages plus haut niveau, permettant de programmer avec des processus légers, comme Java. Ceux-ci devaient être capable de gérer l'exécution en parallèle de plusieurs processus (correspondant généralement à plusieurs applications différentes).

Deux écoles se sont donc longtemps opposées, le modèle préemptif et le modèle coopératif.

3.1.1.1 Le modèle préemptif

Dans ce modèle, dont le meilleur exemple d'utilisation est le système d'exploitation Unix et la norme POSIX, c'est l'ordonnanceur qui décide quand l'exécution d'une tâche doit être interrompue pour laisser place à l'exécution d'une autre tâche. Le programmeur doit donc sécuriser l'accès aux ressources qui risquent d'être utilisées simultanément par les mêmes tâches : il ne peut pas savoir a priori à quel moment de l'exécution la tâche qu'il est en train de programmer sera interrompue. Si celle-ci est justement en train d'écrire dans un fichier ou de modifier une donnée, et que la tâche prenant la main a justement besoin de modifier ou de lire ces mêmes données, les résultats de l'exécution peuvent être pour le moins surprenants...

Pour sécuriser ces accès, on peut par exemple utiliser des systèmes à base de verrous : avant l'accès à une ressource potentiellement partagée, le programmeur d'une tâche place un verrou sur cette ressource, indiquant ainsi à l'ordonnanceur qu'il faut bloquer les autres tâches au point de leur exécution où elles ont besoin d'utiliser cette ressource. Lorsque le programmeur de la tâche décide qu'il n'a plus besoin de modifier la ressource, il supprime le verrou, ce qui indiquera à l'ordonnanceur qu'il peut donner la main à l'une des tâches en attente de cette ressource.

Ce modèle est difficile à utiliser pour le programmeur, qui doit être particulièrement attentif à l'utilisation des ressources partagées, comme le sou-

ligne la littérature à ce sujet. Par exemple, des patrons de conception (*design patterns*) complexes pour la gestion de la synchronisation dans ce modèle peuvent être trouvés dans [90].

De plus, un programme peut sembler fonctionner parfaitement jusqu'au jour où une configuration particulière fait que deux tâches bloquent chacune en attendant que l'autre lève le verrou. Les programmes utilisant ce système sont difficiles à tester, d'autant plus que la plupart des ordonnanceurs ne distribuent pas le temps de calcul entre les tâches de manière déterministe (voir glossaire), ou même simplement reproductible.

Malgré ces inconvénients, c'est le modèle le plus utilisé par les systèmes d'exploitations et par la plupart des langages modernes.

3.1.1.2 Le modèle coopératif

L'approche coopérative a été utilisée par les premiers systèmes d'exploitation grand-public (les premières versions de Windows et de Mac OS utilisaient ce modèle pour faire fonctionner simultanément plusieurs applications).

Dans ce modèle, ce n'est pas l'ordonnanceur, mais la tâche elle-même qui dit quand elle est prête à interrompre son exécution. C'est donc le programmeur qui place l'instruction permettant de donner l'autorisation à l'ordonnanceur d'interrompre son exécution pour reprendre celle d'une autre tâche.

Un des avantages de ce modèle est qu'il présente un modèle de programmation plus simple que le modèle préemptif, l'accès aux ressources partagées entre les différentes tâches n'ayant pas besoin d'être synchronisé par des mécanismes comme les verrous. Il est bien moins facile de bloquer totalement l'application comme c'est le cas avec le modèle préemptif. C'est toujours possible, mais il sera plus facile de détecter le problème.

Un autre des avantages du modèle coopératif est aussi son principal inconvénient : le passage de l'exécution d'une tâche à une autre est coûteuse, car l'ordonnanceur doit procéder à un *changement de contexte*, pour stocker le contexte d'exécution de la tâche qui s'interrompt et restaurer celui de la tâche qui reprend son exécution. Dans le modèle préemptif, le programmeur ne peut pas éviter les changements de contexte inutiles provoqués par l'ordonnanceur, tandis que dans le modèle coopératif, il décide lui-même

du moment où le changement de contexte est nécessaire. Cela signifie qu'en règle générale, on peut écrire des programmes beaucoup plus efficaces avec le modèle coopératif qu'avec le modèle préemptif. Mais cela signifie également que le développeur maladroit peut produire une application particulièrement inefficace en prenant de mauvaises décisions de conception.

Ce modèle a néanmoins été presque complètement délaissé pour la conception des systèmes d'exploitation, car il est généralement considéré que le système est le mieux placé pour déterminer quand il est nécessaire de procéder à un changement de la tâche active, selon les informations dynamiques qu'il possède sur les applications en cours et leurs priorités d'exécution, et surtout parce qu'il n'est pas considéré comme raisonnable qu'une application mal développée pour le système d'exploitation puisse dégrader ses performances.

Cependant, nous considérons que dans le cadre d'un langage haut-niveau ou d'un modèle de *framework*, mis entre les mains de développeurs compétents, les avantages de l'approche coopérative l'emportent sur ses inconvénients.

3.1.2 L'approche réactive synchrone

Cette approche est née de la volonté de fournir un cadre de programmation concurrente *déterministe* et sûr, permettant ainsi de sécuriser le développement d'applications critiques, tels que les systèmes embarqués dans l'aéronautique. Lorsqu'un modèle est déterministe, on peut concevoir facilement des outils permettant d'analyser les propriétés de sûreté et de robustesse d'un programme, comme par exemple de détecter les inter-blocages si courants dans les modèles préemptifs. Dans ce modèle, chaque étape de l'exécution d'un programme est découpé en *instants*. Tous les *événements* se produisant dans l'application au cours du même *instant* sont considérés comme étant simultanés.

Lorsqu'un *instant* se termine, le système *réagit* aux *événements* qui se sont produits au cours de l'*instant* précédent et procède aux calculs spécifiques à l'application. Le temps de cette *réaction* constitue l'*instant* suivant.

Le langage Esterel [11] est un bon exemple de langage utilisant un modèle réactif synchrone. La synchronisation entre les différentes tâches du programme est faite au moyen d'événements que chaque tâche peut émettre,

et que tous les autres composants s'exécutant en parallèle sont capables de capter lors du même *instant*.

La société Esterel-Technologies [33] propose des outils permettant de faciliter le développement d'applications critiques en utilisant ce langage.

L'approche réactive a également été utilisée pour la description du comportement des objets évoluant dans des mondes virtuels, à l'aide de Junior[50], un ensemble de classes Java implémentant un modèle réactif synchrone[21]. De même, la plate-forme InViWo [87] réalisée par Nadine Richard pour la description de comportements d'agents intelligents s'appuie sur le langage Marvin, inspiré par Esterel.

3.1.3 Les *Fair-Threads*

Ce modèle est développé dans le cadre du projet MIMOSA conjoint à l'École des Mines de Paris et l'INRIA, avec notamment une implémentation en Java [20]. Il s'agit d'un modèle équitable, où chaque tâche rattachée au même ordonnanceur obtient un accès égal au processeur. Il existe également une implémentation de ce modèle avec migration pour le langage CAML [98].

3.1.3.1 Un modèle coopératif :

Dans le modèle *Fair-Thread*, un ou plusieurs *fair-schedulers* ordonnancent l'exécution de plusieurs tâches en parallèle. Le modèle d'exécution des tâches (les *fair-threads*) reliées au *fair-scheduler* est coopératif : le programmeur d'un *fair-thread* dispose d'une instruction *cooperate* lui permettant de redonner la main au *fair-scheduler*. Chaque *fair-thread* est ainsi divisé en bloc d'instructions, ces instructions étant exécutées successivement, sans interruption par le *fair-scheduler*.

3.1.3.2 Un modèle inspiré par la programmation réactive synchrone :

Le modèle *Fair-Thread* est inspiré de l'approche réactive synchrone, dont il partage plusieurs aspects :

- la notion d'*instant*, qui correspond ici à chaque bloc d'instruction délimité par les instructions *cooperate* ;
- un modèle de communication entre les tâches par événements, que chaque tâche peut émettre, et qui sont captés par toutes les autres tâches en cours d'exécution au cours du même *instant* ;
- une approche déterministe de la programmation concurrente et un modèle d'exécution pourvu d'une sémantique claire et bien définie ;

Pour obtenir une exécution déterministe des instants, le *fair-scheduler* utilise une stratégie classique appellée *round-robin* («chacun son tour»), qui consiste à exécuter l'instant suivant de chaque *fair-thread* en cours d'exécution dans l'ordre dans lequel ces *fair-thread* ont été rattachés au *fair-scheduler*.

3.2 Processus et difficultés de mise au point

Il n'y a pas si longtemps, on réalisait encore les jeux-vidéo à deux ou trois personnes de manière plutôt artisanale. En même temps que les capacités des ordinateurs personnels et que les bandes passantes disponibles pour les particuliers explosaient, la taille des équipes réalisant un seul jeu-vidéo ont fait de même. De plus, les jeux massivement multi-joueurs sont des applications qui ont une durée de vie supérieure aux jeux de la génération précédente. Il faudra les maintenir et les mettre à jour plusieurs années durant, pendant lesquelles l'équipe en charge de ces opérations peut changer. L'application doit donc posséder de bonnes propriétés de maintenabilité et de lisibilité du code, et ça aussi c'est nouveau, dans le domaine du jeu-vidéo.

Ce passage s'est fait non sans mal, et encore aujourd'hui, le milieu du jeu vidéo a du mal à s'adapter à son statut d'industrie et manque d'outils et de méthodes adéquats.

Malgré le caractère académique d'une thèse et puisque notre but est de proposer des outils permettant d'aider à la réalisation d'applications faisant intervenir un grand nombre de personnes, il nous est apparu nécessaire de décrire quelques méthodologies d'encadrement de projet, afin de mieux situer nos propositions dans le contexte industriel dans lequel elles s'inscrivent.

3.2.1 Quelques processus classiques dans l'industrie

3.2.1.1 Le cycles de vie en cascade et quelques variantes

Cascade : ce cycle de vie du logiciel a sans doute été le premier à être formalisé. On le doit à W.W. Royce qui en a posé les grands principes dès 1970 dans [89]. Ce modèle découpe le cycle de vie d'un logiciel en cinq phases successives (Figure 3.1) : spécifications (identification des fonctionnalités requises), conception (description de l'architecture logicielle satisfaisant les spécifications), codage, tests, et enfin maintenance, une fois que le logiciel est utilisé et que d'inévitables bugs ou demandes de modifications apparaissent. Il est basé sur la constatation empirique, faite à l'époque, que le coût d'un changement du logiciel est multiplié par dix à chaque fois qu'on change de phase. Il vise donc à maîtriser les risques en posant des jalons de vérification après chaque étape. Les étapes sont strictement successives, et un retour à une phase précédente n'est pas envisagé par ce modèle, qui vise justement à s'assurer que le démarrage de chaque phase ne s'effectue que si la précédente correspond strictement au cahier des charges du projet. Ce modèle n'est plus

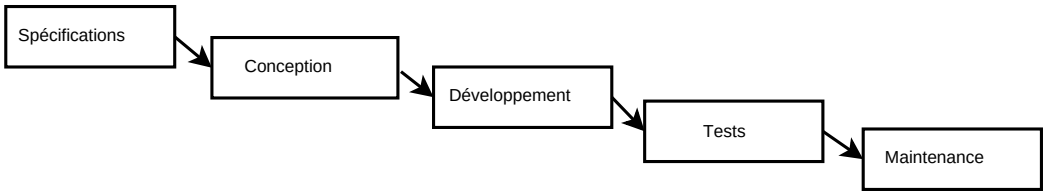


FIG. 3.1 – Cycle de vie en cascade

considéré comme adéquat de nos jours du fait de son manque de souplesse, même s'il convient encore très bien aux petits projets (peu de personnes impliquées, courts dans le temps) dont le cahier des charges est fixe et ne sera pas susceptible de changer. Cette dernière caractéristique devient en effet rarissime dans le développement informatique moderne.

Cycle en V : ce cycle de vie est une variante du cycle de vie en cascade : il partage avec ce dernier le principe théorique de non retour. On ne passe à la phase suivante que lorsque la précédente est achevée. Le cycle de vie est découpé en sept phases successives (Figure 3.2 : spécifications, conception préliminaire (identification des composants logiciels et de leurs interactions),

conception détaillée (description des composants logiciels identifiés), codage, tests unitaires (test des composants logiciels un par un), tests d'intégration (tests de fonctionnement des interactions des composants) et validation (tests finaux, démontrant que chaque fonctionnalité exprimée dans la spécification est réalisée). L'amélioration apportée par ce modèle est la volonté de maîtri-

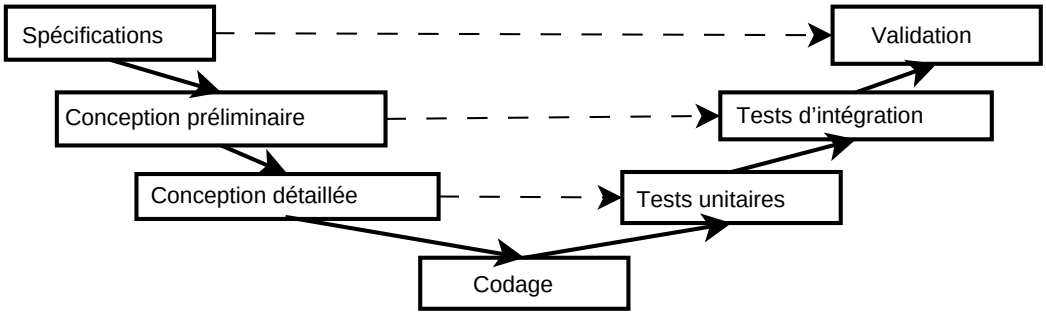


FIG. 3.2 – Cycle de vie en V

ser la qualité du projet, en définissant lors de chaque phase à gauche du cycle les critères et tests qui permettront de valider ultérieurement leurs phases symétriques à droite. En effet, dans une démarche de type cascade, quand le projet prend du retard, l'étape de tests est souvent sacrifiée. L'anticipation des critères de validation de chaque phase à gauche du développement permet de s'assurer que chaque phase sera correctement validée, et si les contraintes de temps rendent nécessaires certains sacrifices durant les phases de tests, il sera possible de les faire de manière intelligente, en privilégiant les fonctionnalités et composants critiques. Cette méthode permet de donner une bonne visibilité sur la qualité du logiciel ainsi produit.

Ce modèle garde cependant le même manque de souplesse que le cycle en cascade par rapport aux modifications. De plus, le suivi de l'avancement du projet est difficile, car il se calcule par phase, et non par objectif de fonctionnalités remplies. Il n'est donc pas très adapté aux gros projets, car il rendra difficile le contrôle de l'avancement et l'évaluation du travail restant à faire.

Cycle incrémental : ce modèle vise à pallier un des inconvénients majeurs du cycle en V : son inadaptation aux gros projets à cause de la difficulté d'évaluer son avancement. Il consiste à découpler le projet en différentes

composantes indépendantes et dont les interactions sont peu nombreuses et bien définies. Le découpage d'un projet en différents composants, en sous-projets, peut parfois être difficile et peu naturel.

Une fois ce découpage effectué, les incréments que représentent chacun des développements peuvent être réalisés certains simultanément, ou successivement (Figure 3.3), suivant les différentes versions du modèle et les dépendances entre les composants identifiés.

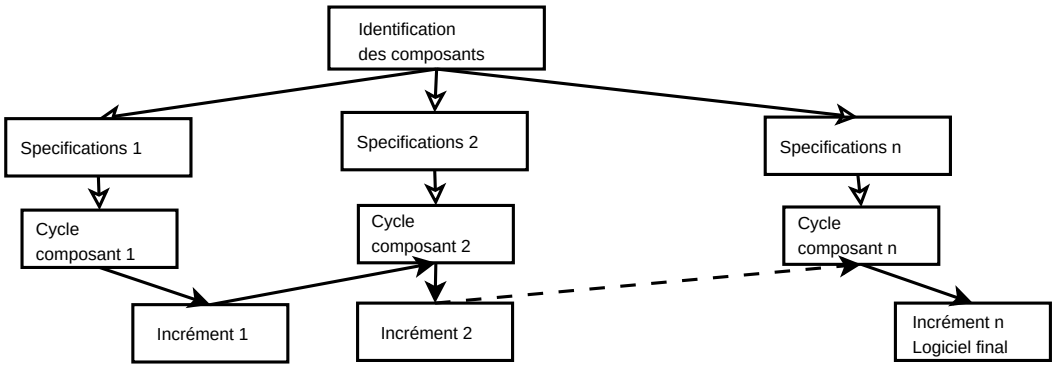


FIG. 3.3 – Cycle de vie Incrémental

Chaque incrément suit un cycle de vie en V ou en Cascade, et est une version stable du logiciel final, terminée du point de vue des composants réalisés, mais incomplète jusqu'au dernier incrément.

Les différents composants ne sont pas nécessairement réalisés dans le même temps, et les spécifications du composant correspondant à un incrément peuvent tenir compte de l'incrément précédent qui correspond à tous les composants précédemment développés, ce qui rend ce modèle un peu plus souple que le cycle en V.

Synthèse : les cycles de vie de la famille des cascades sont adaptés aux projets logiciels très contractualisés, dont le client veut vérifier l'avancement au moyen de jalons.

Ainsi, la plupart des référentiels et normes de développement logiciel qui s'appuient sur ces cycles de vie comportent en plus du projet réalisé un grand nombre de livrables associés, des rapports d'avancement pour chaque jalon, et une documentation très lourde pour chaque étape du cycle (voir par exemple

les standards DOD-STD-2167A [32] ou ISO/IEC 12207 [56], utilisés dans un grand nombre d'industrie du domaine du logiciel critique pour accompagner ces cycles de vie). Même si ce n'est pas prévu par ces cycles, il est très difficile d'empêcher une modification imprévue des spécifications. Le temps pris à réaliser ces livrables du projet nécessaires au passage entre les différentes phases et à donner une visibilité sur son avancement est également accentué par toute demande de modification, qui devra également être répercutée dans toute la documentation associée.

Les processus basés sur ces cycles sont particulièrement mal adaptés aux projets logiciels dont les spécifications risquent de changer à tout moment, que ce soit dû à des changements de spécifications fonctionnelles ou à des problèmes techniques rencontrés en cours de développement. Ils sont également mal adaptés aux projets à risque, dont on ne maîtrise pas bien les aspects techniques. Les projets innovants, faisant intervenir des technologies récentes et difficilement maîtrisables comme les applications Internet et autres applications multi-média ne se développent pas comme des contrôleurs de centrales nucléaires, et les processus encadrant leur développement doivent être plus souples.

En conclusion, les méthodes basées sur ces cycles de vie sont perçues comme trop lourdes pour être utilisées dans la plupart des projets de développement informatique moderne.

Il faut néanmoins les connaître, car ils sont toujours utilisés, en tant que sous-modèles, composants de la plupart des cycles de vie modernes, plus élaborés.

3.2.1.2 Quelques cycles de vie évolutifs

Le principal inconvénient des cycles de vie de type cascade et ses variantes est la difficulté de revenir en arrière à une phase précédente du logiciel, et le fait que les erreurs ou lacunes des premières phases sont découvertes trop tardivement dans le cycle de vie.

Les cycles de vie évolutifs ont pour objectif de résoudre ces problèmes en permettant d'assurer au plus tôt dans le déroulement du projet que le logiciel final correspondra à ce qui est attendu.

Le modèle RAD, *Rapid Application Development* : Ce modèle, principalement conçu pour alléger au maximum le cycle de développement du logiciel a été conçu dans les années 1980 par James Martin, dans le but de proposer des méthodes adaptées à l'encadrement du développement de petites applications graphiques, les méthodes classiques de type cascade étant bien trop lourdes par rapport aux ambitions de la plupart des applications.

Il a été formalisé par l'auteur en 1991 dans [67]. Il s'applique au développement de petits projets faisant intervenir moins d'une dizaine de personnes, développeurs et utilisateur final (qui peut être un client) inclus, ce dernier étant complètement impliqué dans le processus. Cette dernière caractéristique prend à contre-pied les processus classiques de la famille cascade, qui commencent à faire valider un cahier des charges par le client pour revenir le voir seulement une fois le logiciel terminé, avec tous les risques de malentendus que cela comporte. Cette caractéristique rend également cette méthode très ouverte aux modifications de spécifications fonctionnelles en cours de projet.

Il s'appuie sur l'utilisation d'outils de prototypage rapide comme par exemple les outils de génération d'interfaces graphiques, partant du principe que l'achat d'outils fait gagner sur les coûts de développement en raccourcissant les délais. Il consiste en une suite d'itérations perfectionnant à chaque fois le prototype. Les itérations sont très courtes et rapprochées, et chacune fait intervenir l'utilisateur final du produit. Pour garantir leur brièveté, l'équipe de développement peut repousser le développement de fonctionnalités prévues pour le prototype actuellement en construction à l'itération suivante, en décidant avec l'utilisateur final quelles sont les fonctionnalités les moins urgentes à réaliser.

Son principal inconvénient est son manque de passage à l'échelle : la cohérence de l'application étant basée sur une bonne communication des membres de l'équipe, elle est rendue difficile par une équipe qui grossit. De plus, le développement de l'application se fait autour du premier maquettage graphique de l'application, ce qui peut mener à des logiciels difficiles à maintenir car mal conçus lorsque la complexité et la taille du projet augmente.

Cycle en Spirale : Ce cycle de vie itératif, détaillé en 1986 par Barry Boehm [13], a pour but de sécuriser le développement d'une application par

la réalisation d'une spirale de prototypes successifs, approchant petit à petit le logiciel final. La réalisation de chaque prototype peut utiliser un cycle de vie de la famille des cascades.

Chaque itération comprend une nouvelle spécification, suivie par une analyse des points critiques et problèmes techniques ouverts de cette spécification, et enfin d'un prototype démontrant les solutions à ces points critiques et satisfaisant la spécification.

Dans le modèle canonique défini par Barry Boehm, il y a au moins quatre itérations dans ce cycle de développement. Suite à une première analyse des risques du projet et à un premier prototype, la deuxième itération vise à représenter le concept même de l'application, ses idées centrales et éventuellement son interface. La troisième itération sert à définir et valider les spécifications issues de l'étude du deuxième prototype. Puis suit une itération dédiée à la définition et à la validation de la conception générale de l'application. Le prototype issu de cette quatrième étape est le premier prototype opérationnel de l'application.

Après ces étapes canoniques, d'autres itérations peuvent être définies afin d'améliorer certains aspects du logiciel, jusqu'à l'obtention du logiciel final.

A chaque nouvelle itération, le projet peut être re-planifié, avec une nouvelle estimation des coûts et des délais de production. Cette méthode ne convient donc pas aux projets très contractualisés puisque les délais de livraison du produit fini sont réévalués en permanence, mais a l'avantage majeur d'encadrer et de contrôler de manière très sûre le développement de projets pour lesquels le risque est élevé : par exemple, pour des projets utilisant des technologies difficilement maîtrisables ou dont les solutions techniques à certaines fonctionnalités exigeantes sont à inventer.

Processus Unifié : ce cadre a été défini par les «trois amis» Ivar Jacobson, Grady Booch et James Rumbaugh [57] au sein de la société Rational pour encadrer le développement d'applications à l'aide des techniques de modélisation objet UML (Unified Modeling Language) [77]. Une version propriétaire, dotée d'outils simplifiant la mise en œuvre du Processus Unifié est commercialisée par la société *Rational*[2] (maintenant propriété d'IBM).

[2] http ://www-306.ibm.com/software/rational/

Cette méthodologie se veut plus un meta-processus de développement, que l'on peut adapter à ses besoins en retirant les livrables produits par le projet jugés non nécessaires. Il se veut donc léger dans le cadre des petits projets, tout en étant adaptable à des gros projets très contractuels en fournissant les formalismes nécessaires à la production des documents permettant de communiquer au sein de l'équipe de développement et avec le client à l'aide d'UML (Unified Modeling Language) [77].

Ce modèle est centré sur la constitution de l'architecture du logiciel à réaliser. La développement de l'application est guidé par la description de scénarios d'utilisation de l'application (appelés cas d'utilisation), qui seront développés sur le mode incrémental (au même sens que dans le cycle de vie incrémental).

Le Processus Unifié définit quatre phases successives, qui seront réitérées (au même sens que dans les cycles de vie en spirale et itératifs) tant qu'elles n'auront pas donné satisfaction selon des critères définis par la méthode elle-même. La première phase a pour but de vérifier l'intérêt financier du projet, de faire une étude préliminaire des risques, coûts et délais, et de décrire le projet en terme de fonctionnalités clé et de contraintes. La deuxième phase doit fournir la liste des cas d'utilisation, et un prototype décrivant l'architecture du logiciel. La troisième phase consiste à développer les composants permettant de réaliser les fonctionnalités de l'application. Enfin, la dernière phase consiste à faire tester l'application à l'utilisateur final. Le processus définit également neuf types de tâches transverses, qui doivent être prises en compte au cours de chaque phase, comme par exemple la manière d'effectuer des tests ou de communiquer et répercuter un changement de spécification.

Le Processus Unifié déclare donc être le cadre dans lequel tout autre processus de développement doit s'inscrire. Cependant, même lorsqu'on le dépouille de tous les livrables produits par le développement, il nous semble encore bien lourd à mettre en œuvre pour de petites applications telles que celles pour lesquelles la méthode RAD fait des merveilles, et en tout cas plus lourd et complexe que les méthodes agiles que nous allons étudier dans la partie suivante.

3.2.2 Les méthodes agiles

3.2.2.1 Philosophie

Cette famille de méthodes a été consacrée en 2001, lorsque les signataires du *Manifeste pour le développement agile* [3] ont défini les principes devant mener à la création de processus de développement plus adaptés aux applications modernes.

Ces méthodes partagent avec les méthodes évolutives la volonté de maîtriser les risques d'une découverte trop tardive des erreurs ou lacunes des premières étapes de la création d'un logiciel.

Elles ont cependant la volonté d'être plus légères et plus facilement adaptables que le Processus Unifié. Dans le Processus Unifié, un grand nombre de livrables et d'intermédiaires sont définis, et il faut épurer la méthodologie pour l'adapter à chaque projet particulier. Les méthodes agiles partent du principe inverse, en se basant sur la constatation sur le terrain que lorsqu'un projet prévoit de prime abord un certains nombre de livrables en plus de l'application elle-même, il est très rare que la totalité de ces livrables soit produite et mise à jour au long du projet, généralement par manque de temps : elles considèrent donc qu'aucune documentation n'est indispensable a priori, que le temps passé à rédiger de la documentation pour dialoguer avec le client est autant de temps perdu pour le développement de l'application elle même, et préconisent de ne produire des livrables supplémentaires que lorsque c'est absolument nécessaire (requis par contrat, ou indispensable pour la communication dans l'équipe).

Elles préconisent de gérer les risques liés au développement d'applications dont la maîtrise des aspects techniques est délicate ou les spécifications imprécises par l'adoption d'un cycle de vie itératif, comme dans la méthode RAD et le cycle en spirale. Comme dans la méthode RAD également, le client ou un de ses représentant est impliqué jusqu'au cou dans le cycle de vie du développement, et intervient à chaque itération. Un autre point commun avec la méthode RAD est que les itérations doivent être très courtes, afin de pouvoir réévaluer en permanence l'avancement du projet, et les mêmes pratiques de report de fonctionnalités sont mises en œuvre pour s'assurer de leur brièveté.

[3] http ://agilemanifesto.org/

Là où certaines méthodes agiles diffèrent de la méthode RAD, c'est qu'elle ont aussi pour but de s'adapter à de plus gros projets et à de plus grosses équipes de développement. La méthode RAD est selon nous un cas particulier, spécifique à un domaine d'application précis, de méthode agile.

Les cycles de vie sur lesquels reposent les méthodes agiles n'ont rien de nouveau par rapport à ce qui existait auparavant. La seule nouveauté est la constatation qu'il est plus rentable de s'appuyer sur une équipe compétente, et de lui apprendre à communiquer et à travailler efficacement, que de se reposer sur une méthodologie extrêmement rigide, jalonnée et produisant un grand nombre de livrables pour s'assurer que le projet sera mené à bien.

La plus connue de ces méthodes proclamées agiles est sans doute *eXtreme Programming*, présentée par Kent Beck dans [7] qui a scandaleusement fait parler d'elle lorsqu'elle a énoncé ses douze commandements, bien que la méthode *Crystal*, plus récente, plus souple et donc plus facile à mettre en œuvre de manière progressive dans une équipe de développement rodée à d'autres méthodes commence également à émerger [8], même si elle paraît plus adaptée à des petits projets.

3.2.2.2 eXtreme Programming

Nous allons ici détailler un peu plus les principes généraux d'*eXtreme Programming*, afin de montrer comment cette méthode propose de remplir les objectifs fixés par le *Manifeste pour le développement agile*.

Cette méthode, qui par bien des aspects est plus une discipline de développement, s'appuie sur douze pratiques, afin d'assurer quatre valeurs centrales :

- la **communication** permanente entre le client et l'équipe de développement : le premier fait idéalement partie intégrante de l'équipe ou est du moins à portée de main et décide des priorités des fonctionnalités à réaliser. La communication est continue également au sein de l'équipe, qui doit à tout moment avoir la même vision de l'application, et dont aucun individu ne doit être irremplaçable ;
- la **simplicité** des éléments produits par le projet : les développeurs reviennent en permanence sur le code déjà produit pour le simplifier et le minimiser en regard des nouveaux développements (cette pratique est appelée *refactoring* [43]), et la production d'autres livrables est limitée

au strict nécessaire.
- des **retours fréquents**, continuels, sur le développement du projet, à travers notamment des tests unitaires de non régression répétés, et des itérations très courtes, durant entre une et quatre semaines.
- et enfin, le grandiloquent **courage** d'admettre ce qu'on peut ou ne peut pas réaliser en temps et en heure, afin d'avoir toujours une planification réaliste.

Là où le Processus Unifié et la méthode RAD sont des processus de développement guidés par la description de scénarios d'utilisation, un projet eXtreme Programming est guidé par la description des tests : le client écrit les tests de vérification d'une fonctionnalité avant que celle-ci ne soit développée. De même, les développeurs écrivent les tests unitaires avant même de coder.

Les douze pratiques préconisées par eXtreme Programming et définies dans [7] sont alors les suivantes :

1. Le jeu du planning : il consiste à planifier juste après chaque itération le contenu et la durée de l'itération suivante, en tenant compte des priorités fixées par le client et des estimations des difficultés techniques.
2. Petites livraisons : un exécutable est livré au client à chaque itération, pourvu à chaque fois de nouvelles fonctionnalités.
3. Métaphores : un schéma, diagramme ou une description littéraire très brève doit permettre de représenter l'application assez simplement pour que toute l'équipe aie la même vision du développement. La métaphore utilisée doit être compréhensible par tous, nécessiter pas ou peu de maintenance au cours du déroulement du projet.
4. Conception simple : garder une conception de l'application la plus simple possible. Continuellement traquer et simplifier les aspects complexes.
5. Tests : le client écrit les tests pour les fonctionnalités, les développeurs écrivent les tests unitaires avant de coder, et ces tests sont effectués régulièrement pour vérifier la non-régression de l'application.
6. Remaniement : utiliser régulièrement les techniques de *refactoring* pour enlever le code dupliqué, le code mort, et le simplifier.
7. Programmation en binôme : les programmeurs travaillent en équipe de deux, l'un écrit le code, pendant que l'autre vérifie sa correction et sa lisibilité, et les choix techniques sont effectués en commun. Cette pratique

qui semble scandaleuse et peu gratifiante est très efficace pour garantir un codage homogène compréhensible par toute l'équipe et maintenable sur le long terme.

8. Propriété collective : tout le monde est capable et peut si nécessaire modifier n'importe quelle partie du code de l'application.
9. Intégration continuelle : dès qu'une tâche d'implémentation est terminée et testée, on procède à l'intégration, même plusieurs fois par jours selon les projets.
10. Semaine de quarante heures : comme un développeur fatigué travaille moins bien, les heures supplémentaires ne sont pas autorisées deux semaines de suite.
11. Client sur place : un client ou un représentant du client est disponible à plein temps, pour aider à concevoir le logiciel, répondre aux questions, et écrire les tests d'intégration.
12. Standards de code : les programmeurs adoptent et respectent à la lettre un standard de code. Ce standard bien respecté, il doit être impossible de distinguer qui a écrit quelle partie de l'application.

Les principaux avantages de la méthode eXtreme Programming sont qu'elle permet de fournir une application facilement maintenable car bien écrite, de manière homogène et compréhensible et également d'éviter qu'un des développeurs ne devienne si indispensable que son départ constitue un gros risque pour le projet. Pourtant, et malgré le principe de la programmation en binôme qui peut sembler de prime abord humiliant pour un développeur expérimenté, eXtreme Programming et les autres méthodes agiles sont les processus qui donnent le plus d'importance au travail des développeurs, laissant à leur charge la qualité de l'application, les re-planifications permanentes, et éliminant au maximum les livrables supplémentaires à produire au cours du projet.

Le principal inconvénient d'eXtreme Programming est qu'il est difficile d'obtenir la coopération du client, qui préfère en général avoir un contrat en béton armé définissant la livraison de toutes les fonctionnalités du projet qu'il a défini au tout début à une date donnée, et ne pas avoir à s'occuper de ce qui se passe entre la signature et la livraison.

Ces méthodes sont donc de fait plus adaptées lorsque le client, l'utilisateur final, fait partie de la société. Dans le domaine du jeu vidéo, le *game-designer* peut très bien jouer le rôle du client dans le cadre de ces méthodes.

3.2.3 Processus de développement d'un jeu massivement multi-joueurs

Le livre écrit par Jessica Mulligan et Bridgette Patrovsky [72] est une source assez représentative du processus industriel qui mène à la réalisation d'un jeu massivement multi-joueurs (ou de ce qu'il devrait être dans l'idéal selon les experts, tout du moins). Nous allons ici commenter certaines étapes de ce processus.

3.2.3.1 La conception fonctionnelle et technique

Les auteurs recommandent un développement qui suit un cycle de vie qu'on peut rapprocher du cycle incrémental, et pour faciliter et sécuriser le développement, recommandent l'utilisation ou le développement préalable d'outils logiciels dédiés d'un haut niveau d'abstraction, comme des moteurs d'intelligence artificielle, graphiques ou réseau, dédiés à la gestion des différentes fonctionnalités du jeu. La conception et le développement du *game-play* lui même suivent un cycle de vie de style cascade.

Avant que quoi que ce soit ne soit implémenté, il est recommandé de passer par une étape cruciale de conception. La conception des mécanismes, des règles du jeu et la conception technique sont réalisées en parallèle dans le même temps, et les concepteurs dans chaque domaine doivent collaborer étroitement : dans un jeu massivement multi-joueurs, il y a des interactions qui ne peuvent pas être réalisées techniquement, qui requièrent de considérables investissements en solutions d'hébergement et de déploiement, ou qui seront très délicates à mettre au point. Le but de cette première étape est de concevoir le *game-play* du jeu, dans ses aspects à la fois techniques et fonctionnels.

Pour que cette première étape soit un succès, il n'y a actuellement pas d'autre support que de bonnes pratiques de conduite de projet. Cependant, le travail créatif qui consiste à concevoir les aspects fonctionnels (les règles et la mécanique) d'un jeu demande des compétences très différentes de celui qui consiste à concevoir le logiciel lui-même, en tenant compte des aspects techniques complexes qui y interviennent (réseau, concurrence, passage à l'échelle). Ce sont des domaines d'expertise totalement différents. Pour que la communication soit efficace, il faut souvent l'intervention d'un concepteur

fonctionnel avec un solide bagage technique ou à l'inverse, d'un technicien ayant de bonne notions et une bonne expérience dans la conception de jeu.

De plus, même si l'équipe de développement comporte un de ces précieux spécimens, ou en engage un pour la durée de cette première phase, ce que dicte l'expérience se résume souvent à : «ce qui a marché pour un jeu précédent devrait marcher cette fois encore».

3.2.3.2 Le *beta-test*

La dernière étape de la réalisation d'un jeu, juste avant que le jeu ne soit déployé, est appelé le *beta-test*, et consiste à faire tester le jeu par des utilisateurs. C'est une étape héritée de l'histoire du développement de jeux-vidéo, datant de l'époque où les jeux étaient développés en peu de temps par une équipe réduite.

Après un *alpha-test* réussi qui consiste à tester le jeu à une petite échelle sur un réseau fermé et sans essayer de passer à l'échelle en nombre de joueurs, le jeu est testé par des joueurs dans des conditions réalistes de déploiement. Le principe du *beta-test* est de pousser le jeu à ses limites, en terme de performances et de ressources matérielles. Cette phase est aussi celle où les conséquences des inévitables réactions imprévisibles des joueurs vont être observées. En bref, c'est l'étape lors de laquelle les conséquences d'une conception insuffisante à faire correspondre les aspects fonctionnels et techniques vont se révéler.

3.2.3.3 Critique de ce mode de production

Entre les deux phases précédentes, il peut s'être passé des mois et parfois des années de développement très coûteux (le développement d'un jeu massivement multi-joueurs fait intervenir des équipes de tailles conséquentes). Le coût d'un retour en arrière pour ajuster les solutions techniques au *game-play*, ou pour revoir le *game-play* par rapport à des problèmes majeurs de *jouabilité*, peut être suffisant pour causer la banqueroute du projet.

Un bon moyen de minimiser ce genre d'erreur serait de réaliser un prototype pour tester la conception technique des interactions, et de l'utiliser conjointement avec des outils de simulations de conditions critiques de

ressources matérielles (perte de message réseau, augmentation de latence, consommation de CPU...). Une telle approche permettrait d'aider à découvrir plus tôt dans le déroulement du projet quelles recettes techniques peuvent permettre de réaliser les fonctionnalités voulues. Cette approche permet également d'encourager les projets de jeux innovants du point de vue des interactions possibles, en démontrant précisément, lors de la première étape de conception, quelles fonctionnalités sont faisables, et d'observer si elles ont une bonne qualité de jouabilité dans des conditions réelles. Même s'il peut être difficile de simuler ou de prédire le comportement des joueurs lorsque le jeu sera enfin terminé (les endroits du monde virtuel qu'ils choisiront pour se rassembler ou pour se battre, et donc les machines du serveur correspondant dans certains choix d'architecture par exemple), le fait de réaliser un prototype réduit grandement le risque d'un *beta-test* désastreux.

Cependant, le prototypage a encore une très mauvaise image dans l'industrie du jeu vidéo, parce qu'un bon prototype prend du temps, et parce que la plupart du temps, le code produit n'est pas exploitable après coup. Effectué souvent trop vite et dans l'urgence à des simples buts de démonstration, il manque de réutilisabilité. C'est donc considéré comme une perte de temps.

3.3 SCOL

Il existe sur le marché quelques outils qui proposent de faciliter l'intégration de jeux massivement multi-joueurs. Il nous a cependant semblé impossible de les évaluer objectivement dans le cadre de ce travail académique, à cause d'un manque de visibilité sur leurs capacités : ils n'ont que très rarement été utilisés dans l'industrie à ce jour. Le lecteur curieux peut trouver une liste de ces outils dans [54], et un descriptif des trois principaux dans [39].

Il est cependant impossible à l'auteur de ne pas mentionner dans le cadre de ce chapitre la technologie SCOL de ses anciens employeurs, qui est après tout le facteur déclenchant de cette thèse. Il s'agit d'une solution passée en *open-source* depuis le dépôt de bilan de la société qui en était propriétaire.

La technologie SCOL (pour *Standard Cryo On-Line*) a été créée par Sylvain Huet pour la société Cryo-Networks, pour permettre de réaliser des applications Internet et multimédia, dont le dénominateur commun est la

représentation des utilisateurs connectés dans un espace en trois dimensions où ils peuvent se rencontrer et interagir. Parmi les réalisations de la société et de ses partenaires, on peut citer des communautés virtuelles comme Cryopolis, des sites marchands et des jeux multi-joueurs en ligne comme *Venise*, *Fog*, et *La chasse au trésor*. Peu avant son dépôt de bilan en Juillet 2002, la société finissait de mettre au point un jeu en ligne massivement multi-joueurs à l'aide cette technologie.

3.3.1 Le langage SCOL et sa machine Virtuelle :

La technologie SCOL est basée sur une machine virtuelle, qui exécute des programmes écrits dans le langage SCOL. Cette machine virtuelle est caractérisée par une gestion automatique de la mémoire (elle possède un *garbage collector*) et une gestion des communications réseau (la façon dont celles-ci sont prises en charge est en effet masquée par le langage). Une application SCOL est donc un ensemble de programmes écrits dans ce langage, que l'on donne à compiler (à la volée) et à interpréter à une ou plusieurs machines virtuelles communicantes, éventuellement distantes.

L'approche est contraire à celle de VRML (*Virtual Reality Modeling Language*) [100] : alors que cette dernière repose sur une immersion dans un espace en trois dimensions, à laquelle on superpose la notion de cohabitation des utilisateurs connectés, le cœur même de la technologie SCOL est le modèle de communication réseau auquel s'ajoutent les bibliothèques dédiées à la simulation d'un univers. Les plus importantes sont le moteur 3D et des bibliothèques 2D, son et SQL.

SCOL plonge ses racines dans la longue tradition des langages fonctionnels, nés au sein de la communauté académique à la fin des années 50. C'est un langage applicatif, polymorphe paramétrique avec inférence de types et qui manipule des valeurs fonctionnelles. En fait, SCOL a été directement inspiré par Caml [63] dont il adopte une grande partie de la syntaxe et l'inférence de types, et a voulu profiter de bon nombre des avantages de son grand-frère : il se devait donc d'être un langage d'un haut niveau d'abstraction, mettant à profit le polymorphisme paramétrique pour une meilleure réutilisabilité du code produit, et le typage statique pour détecter dès le chargement les incohérences éventuelles et permettre une compilation plus efficace.

Le noyau du langage SCOL se présente comme un langage simple, que l'on peut expliquer en quelques pages. Contrairement aux langages impératifs et orientés objet généralement adoptés par l'industrie, SCOL évite au programmeur le besoin d'écrire explicitement les types des variables et des fonctions sans pour autant mettre en danger l'exécution des applications.

Nous avons décrit la sémantique opérationnelle du langage SCOL dans [14].

3.3.2 Packages et chargement dynamique

Le chargement dynamique est un grand avantage pour la réalisation d'un monde virtuel : lorsqu'un utilisateur se connecte, il n'a pas nécessairement besoin dès le départ de toutes les fonctionnalités de l'application, et peut donc se contenter de télécharger depuis le serveur dans un cache le code source correspondant à un minimum de fonctionnalités indispensables, puis de le charger dans la machine virtuelle (ces deux opérations étant prises en charge de manière transparente par l'application). Au fur et à mesure de sa visite, il se procurera, au besoin, le code SCOL nécessaire à ses activités : tout se passe exactement comme lorsqu'on *surfe* de page en page sur Internet.

Le *package* est l'unité de chargement de la machine SCOL. C'est un fichier contenant une suite de définitions de types, de variables, et de fonctions. Le chargement dynamique d'un package permet de modifier l'environnement d'exécution en y ajoutant de nouvelles définitions ou en masquant les anciennes. L'originalité de la machine SCOL est qu'elle sait gérer plusieurs environnements, qui ne sont pas forcément indépendants : ceci permet de partager certaines variables ou fonctions entre des environnements différents.

3.3.3 Canaux et messages

Autre originalité de la machine SCOL, l'association d'un environnement avec une liaison réseau, que l'on appelle *canal*. Il est néanmoins possible de définir des canaux sans liaison réseau associée.

Le modèle d'exécution de SCOL repose fondamentalement sur cette notion : l'évaluation d'une expression s'effectue toujours dans le contexte d'un canal donné. La liaison réseau prend la forme d'une connexion TCP ou UDP

avec une autre machine virtuelle. Dans le cas simple d'une application qui n'est pas liée au réseau, la connexion est vide et on se trouve alors dans un modèle d'évaluation non distribué. Si la connexion est définie, on se trouve dans le cas plus général d'une application SCOL répartie qui permet l'activation de fonctions définies dans des environnements potentiellement distants.

L'architecture de communication de la machine SCOL est fortement basée sur un modèle client-serveur. Afin que deux machines SCOL puissent communiquer, il faut que l'une d'entre elles adopte le rôle du serveur, et que l'autre ouvre une connexion vers ce serveur. Deux primitives très simples permettent d'initier ce dialogue. Une fois la connexion établie via la construction de canaux dédiés au dialogue, la communication peut se dérouler entre les deux machines, par envoi par l'une de *messages* faisant partie du vocabulaire de l'autre. Pour déclencher une fonction distante, il faut en plus de la convention de nommage que le message envoyé soit défini en déclarant les types des arguments de cette fonction. Ce mécanisme s'apparente à un système de RPC (*Remote Procedure Call*, l'ancêtre de RMI, voir partie 2.3.2) simplifié, où seuls des paramètres de type entier et chaîne peuvent être communiqués et où l'émetteur n'a aucune information sur le succès ou l'échec de l'appel.

3.3.4 Système de modules distribués et outil d'intégration

La bibliothèque DMS (*Distributed Module System*) et l'outil d'intégration associé SCS (*Site Construction Set*), écrits en SCOL, ont pour vocation de permettre d'assembler simplement des briques de base, implémentant des fonctionnalités de haut niveau, et de mettre ainsi la réalisation d'un monde virtuel à la portée d'un public intégrateur non spécialiste du domaine. DMS repose sur une architecture client-serveur, et utilise les protocoles TCP et HTTP pour faire communiquer les différents hôtes de l'application répartie.

Les briques de base d'une application réalisée à l'aide du SCS sont des programmes d'un haut-niveau d'intégration, appelés *modules*, et réalisant chacun une fonctionnalité fréquemment rencontrée dans un monde virtuel : par exemple, un *chat*, un mécanisme d'authentification, ou tout simplement un rendu 3D permettant de visualiser les avatars des utilisateurs connectés au même monde. Ces modules, dans le SCS, se présentent comme de simples

boîtes noires, munies d'entrées (des fonctions particulières implémentées par le module et déclarées comme *actions*), de sorties (qui sont des *événements* produits par le module) et éventuellement d'une interface utilisateur. L'intégrateur assemble les modules via le SCS, en reliant les entrées des uns aux sorties des autres, et définit l'interface utilisateur de son site en combinant entre elles les interfaces des différents modules qu'il utilise.

L'utilisateur du SCS n'a pas à se soucier de la répartition de l'application : les modules peuvent avoir une partie cliente et une partie serveur, la gestion de la distribution étant donc essentiellement réservée au développeur qui prend en charge la communication entre les parties clientes et serveur du module. Une fois le site assemblé, les communications générées par les liens tissés par l'intégrateur (un événement et une action liés pouvant indifféremment être client ou serveur) sont prises en charge par le *framework* DMS.

3.3.5 Critique

Nous avons décrit dans [18] les principales lacunes de la technologie SCOL pour la réalisation d'un jeu massivement multi-joueurs.

En SCOL, il n'existe aucun moyen autre que le *package* et la fonction pour structurer le code[4], et les limitations de ce modèle sont vite atteintes lorsqu'on s'attelle au développement d'une application importante. Aucun mécanisme du langage SCOL ne permet de définir des structures de données abstraites, car il n'y a pas de mécanisme d'encapsulation. Bien sûr, il est toujours possible, avec de la rigueur et des conventions de nommage, de venir à bout de ces difficultés, mais ces stratégies de conception sont vouées à l'échec : elles n'existent souvent que pour ne pas être respectées. De plus, en terme de génie logiciel, l'organisation des programmes a également pour but de permettre de réutiliser le code et de réaliser des tests unitaires. Or, nous l'avons vu dans la présentation du langage SCOL, la sémantique d'un *package* est fortement liée à celle de son environnement, les liaisons étant résolues dynamiquement. Il est donc très difficile de décrire le comportement

[4]Les modules de DMS ne sont pas des entités du langage, mais des constructions abstraites, d'un haut niveau d'intégration fonctionnelle, du *framework*. Ils peuvent souvent atteindre une taille de code suffisament respectable pour nécessiter eux-même une structuration plus élaborée.

d'un *package* sans tenir compte de son contexte et de tirer profit d'un code préexistant.

De plus, alors même que de grands progrès en ce sens ont été effectués pour les langages fonctionnels, il n'y a pour SCOL aucun outil permettant de mettre au point du code : ni *debugger* digne de ce nom, ni *profiler*. Il est vrai que les caractéristiques de SCOL ne facilitent pas la construction de ces outils (la perte des informations de type à l'exécution complique la construction d'un *debugger*, même si ce n'est pas impossible à réaliser). Ce sont pourtant les outils indispensables à la réalisation d'applications importantes, car ils permettent de découvrir le plus tôt possible dans le cycle de développement les éventuelles failles d'une application, et donc de gagner du temps à terme : sans *profiler*, il n'est pas rare de ne découvrir qu'une fonctionnalité critique est inefficace qu'au moment de la mise en service de l'application, d'autant plus que si celle-ci est de type monde virtuel, la faiblesse ne sera souvent découverte qu'à partir d'un certain nombre de clients connectés.

Le langage SCOL s'est révélé très efficace pour le développement d'applications de type communautés virtuelles à petite échelle, et ceci grâce à son expressivité, au chargement dynamique, et à certaines autres caractéristiques propres à sa famille de langages, dont les plus évidentes sont sans doute la sûreté du typage statique et l'inférence de type. De plus, contrairement à ce qui est reproché à beaucoup de langages fonctionnels [101], SCOL était muni de toutes les bibliothèques facilitant son utilisation pour les applications visées.

La façon dont la technologie a été pensée ressemble plus à la vision d'un Internet communautaire en 3D, dans lequel on se déplacerait, en rencontrant les autres utilisateurs, d'un petit monde virtuel à l'autre, qu'à celle d'une application massivement multi-utilisateurs ayant des impératifs de performances. Les problèmes de facteur d'échelle qui se posent lorsqu'on cherche à réaliser une application telle qu'un jeu massivement multi-joueurs n'ont pas été pris en compte dans la conception initiale de la technologie. SCOL manquait également d'un véritable environnement de développement, ainsi que de constructions du langage propres à faciliter l'exercice du génie logiciel : s'il est possible de s'en passer lorsque les projets sont peu importants, il devient extrêmement difficile de contrôler le développement d'un projet impliquant plus d'une demi-douzaine de développeurs si on ne dispose pas de tels outils.

Chapitre 4

Game-design et conception technique

4.1 Introduction

Ce chapitre a pour but de faire une synthèse des problèmes que nous cherchons à résoudre, en regard de l'état de l'art présenté lors des précédentes parties de ce manuscrit. Nous y décrivons la réflexion qui nous a amené à élaborer les solutions proposées dans le cadre de cette thèse. Nous expliquons pourquoi les outils et *frameworks* actuels ne suffisent pas, et comment nous proposons de réaliser un environnement de développement dédié à la réalisation de jeux massivement multi-joueurs.

Nous défendons l'idée de la nécessité de fournir aux développeurs un *framework* muni d'une sémantique simple et bien définie, permettant ainsi l'adjonction ultérieure d'outils de mise au point, et nous expliquons pourquoi les méthodologies basées sur le raffinement de prototypes successifs sont les seules solutions viables pour mettre au point une application si complexe.

Nous décrivons ensuite les grandes idées sur lesquelles repose le modèle défini au cours de cette thèse, le comparons avec des approches voisines, et nous illustrons son utilisation par des exemples simples.

4.2 Le problème des interactions dans un monde virtuel

4.2.1 L'importance de la qualité des interactions dans le *game-design*

Le cœur d'un jeu en ligne ou d'un monde virtuel, c'est l'interaction. On joue pour interagir avec d'autres joueurs, dialoguer, collaborer ou se battre, commercer et échanger des objets du monde virtuel. Le fait de déplacer son avatar dans un monde virtuel est une interaction du joueur qui modifie l'état du monde virtuel en modifiant sa position. Le fait d'acheter un objet virtuel dans un magasin du monde virtuel en est une autre. Lorsque des joueurs se battent, ils interagissent les uns avec les autres tout en modifiant également l'état du monde...

Ce sont les interactions que le joueur a avec les autres joueurs et avec le monde persistant lui-même qui rend l'expérience si différente des jeux traditionnels.

Cependant, pour que le sentiment d'immersion dans le monde virtuel soit réussi, ces interactions doivent sembler les plus naturelles possible. Pour une bonne expérience de jeu, leurs propriétés techniques doivent être adaptées au *game-play*. Par exemple, nous avons comparé au chapitre 1 les manières dont se déroulent les combats entre joueurs, dans un FPS comme Quake[84] ou Half-Life[51], et dans un jeu massivement multi-joueurs comme Everquest : un FPS se joue beaucoup sur la qualité des réflexes des différents participants et requiert une propagation très rapide du moindre mouvement d'un joueur aux ordinateurs des autres protagonistes, tandis que dans Everquest, c'est le nombre d'informations à propager aux participants qui devient très vite important et la précision et la rapidité de transmission des positions ne sont plus aussi cruciales.

Pour le développeur, définir une interaction dans un jeu massivement multi-joueurs, c'est définir de quelle manière un joueur, ou un événement contrôlé par le serveur, modifie l'état global du monde virtuel. Nous avons vu dans le chapitre 2 un échantillon représentatif des techniques actuellement utilisées pour développer un jeu en ligne compte tenu des impératifs de jouabilité et d'immersion du joueur dans la monde virtuel. Nous allons main-

tenant expliquer pourquoi, malgré le fait que ces techniques soient connues, il est très délicat de faire les choix qui permettront de réaliser un jeu satisfaisant les qualités des interactions requises par le *game-play*.

4.2.2 La délicate mise au point d'un jeu en ligne

Nous allons dans cette partie faire une synthèse des difficultés rencontrées lors de la réalisation d'un jeu massivement multi-joueurs, étudiées lors des chapitres précédents. Ce retour sur les points clés du développement d'une telle application a pour but d'étayer notre argumentation.

4.2.2.1 Les limites physiques des interactions

Un jeu massivement multi-joueurs est une application distribuée sur Internet qui doit faire partager une même vision du monde à tous ses participants. Comme une consistance absolue de l'état sur tous les participants au monde virtuel n'est pas possible compte tenu de la nature de l'Internet, le problème consiste à déterminer pour chaque état local à quel point la synchronisation avec sa valeur dans l'état global de l'application est nécessaire. La gestion de l'état global est le principal problème de mise au point pour les jeux massivement multi-joueurs.

Par exemple, un calcul assez simple montre que dans un réseau parfait, où les paquets d'information circulent à la vitesse de la lumière, un aller-retour entre Paris et Melbourne dépasse le temps considéré comme étant celui du réflexe humain (100ms). Il y a donc une limite physique à la rapidité de la synchronisation. Nous avons étudié des techniques permettant de pallier ce problème. Mais elles sont difficiles à mettre en œuvre car chacune présente des inconvénients par rapport à une autre caractéristique de l'application, et sont donc à utiliser au cas par cas, selon le *game-play* désiré.

4.2.2.2 La difficulté de tester les interactions en conditions réelles :

De plus, la solution au compromis précédemment cité est difficile à mettre au point car la qualité de l'interaction réalisée ne sera vraiment testée en conditions réelles que dans les dernières étapes du développement. Dans le

contexte d'un jeu en ligne, certaines faiblesses de conception n'apparaissent que lorsque le jeu est déployé dans des conditions réelles d'exécution, qui doivent tenir compte de la réalité d'Internet avec sa latence imprévisible, et de la participation d'un grand nombre de joueurs connectés afin de vérifier le passage à l'échelle. Ces conditions s'obtiennent généralement seulement lors du *beta-test*, qui est la toute dernière étape avant le lancement du jeu. Anarchy Online, un des premiers jeux massivement multi-joueurs a succès a bien failli ne pas se remettre de problèmes découverts seulement après la mise en ligne de l'application [47]. Toléré par les joueurs à l'époque ou les jeux en ligne étaient encore un type d'application novateur et la communauté confidentielle, ce type de lancement n'est plus aujourd'hui considéré comme acceptable, le grand public étant désormais plus exigeant. Un jeu dont le *beta-test* est désastreux se fera non seulement très vite une mauvaise réputation, mais devra provoquer un remaniement de toute l'application, et donc probablement des délais de lancement retardés et des coûts supplémentaires potentiellement élevés.

4.2.2.3 Le besoin de spécialiser finement chaque interaction :

Pour complexifier encore le problème, nous avons vu que la qualité des interactions des joueurs avec le monde virtuel est fortement liée à un style de *game-play* : selon le type de jeu, chaque modification de l'état du jeu doit donc être répercutée de manière différente, en faisant les compromis adéquats compte tenu des impératifs de fiabilité, de sécurité ou de rapidité de chaque propagation d'événement.

De plus, dans un même jeu, les différents états composant l'état global du jeu peuvent requérir différentes qualités d'interactions et donc différents compromis. La description des interactions doit donc également s'effectuer au cas par cas au sein d'un même jeu.

Mettre au point les interactions d'un *game-play* se résume donc à résoudre un compromis entre les qualités requises pour chaque caractéristique de l'application.

4.2.3 Les jeux massivement multi-joueurs actuels

Aujourd'hui, les types d'interactions et de *game-play* réalisables sont bien connus des concepteurs de jeu, et le lancement d'un jeu massivement multi-joueurs est moins dangereux, surtout si on se contente d'utiliser les recettes déjà éprouvées. Cependant, cela mène à une ressemblance frappante entre les jeux à succès, ennuyeuse pour les joueurs. Cette ressemblance est particulièrement flagrante du point de vue des interactions possibles entre les joueurs et le monde virtuel. Les jeux à succès les plus récents, comme *Starwars Galaxies* [95] ou *World of Warcraft* [53], ne proposent aucune innovation dans la manière dont les joueurs interagissent par rapport à *Everquest*, le pionnier du genre.

Neocron [76] est un exemple de jeu massivement multi-joueurs ayant toutefois essayé d'innover en terme d'interactions. Son lancement a d'ailleurs plus ressemblé à celui des premiers jeux massivement multi-joueurs, avec de nombreux tâtonnements dans les premiers temps, et son arrivée à maturation s'est faite seulement après des mois de jeu, certains problèmes n'ayant même jamais été réglés. Ce jeu a désormais été interrompu, les développeurs repartant de presque zéro pour en développer une seconde version sur des bases saines.

On peut donc penser que les concepteurs de jeu ne manquent pas d'envie d'innover et de nous proposer de nouveaux types d'interactions dans les jeux en ligne massivement multi-joueurs. Cependant, l'importance des budgets nécessaires et la difficulté de mise au point de nouveaux styles d'interactions freinent actuellement l'innovation.

4.2.4 Les outils dans l'industrie :

Depuis l'explosion du marché des jeux en ligne, de nombreuses solutions ont vu le jour dans l'industrie, pour aider au développement des jeux en ligne. On peut trouver une étude de ces solutions dans le rapport annuel 2003 de l'*International Game Developers Association*(IGDA) [54] sur les jeux en ligne. Sans donner une liste complète et une critique de ces solutions commerciales, on peut néanmoins les ranger en trois catégories principales selon ce qu'elles permettent de réaliser :

– les moteurs réseau et moteurs de jeux génériques, conçus pour le déve-

loppement d'un jeu particulier et éventuellement raffinés pour devenir plus génériques ;
- les solutions plus ou moins complètes de sociétés qui vendent juste des technologies, comme des moteurs d'Intelligence Artificielle divers et variés, permettant la génération de scénarios interactifs, la modélisation de comportements intelligents pour des agents ou des moteurs physiques ;
- des moteurs pour la réalisation de jeux classiques, adaptés après-coup pour satisfaire les besoins d'un passage à des versions multi-joueurs.

À ce jour, pratiquement tous les projets de développement de jeux massivement multi-joueurs ont commencé par une phase de développement d'outils et de bibliothèques adaptés à chaque jeu. Les composants logiciels développés lors de ces phases ont très rarement été utilisés pour des produits commerciaux différents de celui pour lequel ils ont été réalisés : conçus avec un projet de jeu particulier à l'esprit, ils manquent de généricité, et sont souvent d'un niveau d'abstraction fonctionnelle trop élevé pour permettre un réglage fin des interactions pour modifier celles qui sont prévues pour le jeu initial.

Les recettes techniques et les modèles de communication évoluent constamment, sont comme nous l'avons vu fortement liées au choix de *game-design*, mais sont généralement, malgré tout, parties intégrantes de ces solutions et donc difficilement modifiables ou paramétrables. Ces solutions peuvent être utilisées avec succès pour faire des jeux dont les principes et interactions ne diffèrent pas de trop de ceux pour lesquels elles ont été créées. Un jeu innovant du point de vue des interactions (et donc du *game-play* qu'il propose) ne peut donc pas tirer profit des outils existants.

Un grand nombre des sociétés s'intéressant à ce marché proposent plutôt l'infrastructure nécessaire à l'exploitation des jeux en ligne (par exemple, login sécurisé, services de mise à jour du jeu, services de discussion en ligne par mode texte entre les joueurs). Cette famille de solutions ne fournit aucune solution en terme de modélisation des interactions. Enfin, d'autres sociétés essaient de fournir des architectures de communication distribuées, un découpage en différents services des serveurs gérant le monde virtuel. Mais la plupart n'ont pas fait la preuve de leur généricité et de leur adaptation dans le cadre du développement d'un jeu massivement multi-joueurs [54]. De plus, nous avons vu que chaque architecture a un impact sur la qualité des caractéristiques du jeu, et nous doutons donc qu'une solution générique puisse être trouvée par ce biais.

L'outil idéal pour permettre de modéliser de nouveaux types d'interaction pourrait être quelque chose de similaire dans ses principes de base aux solutions industrielles qui existent pour le développement de jeux-vidéo traditionnels. Par exemple, Criterion Software propose Renderware [86], un ensemble d'outils et de bibliothèques couvrant le savoir-faire actuel dans ce domaine. Cette solution est ouverte, ce qui signifie que de nouvelles bibliothèques peuvent être ajoutées par les partenaires de l'industriel. Virtools dev [99] en est un autre exemple, et propose un kit de développement comportemental qui permet d'utiliser un certain nombre de briques de base à assembler à l'aide d'un langage à syntaxe graphique, ainsi qu'un langage de script et les moyens d'intégrer de nouvelles briques à la solution. Cette solution fournit également des facilités utilisables par un concepteur de jeu pour tester le logiciel ainsi produit.

Une approche aussi ouverte, incluant outil d'intégration et outils de tests, et ne faisant aucune hypothèse sur le modèle de communication et les modèles de données serait intéressante à utiliser pour le développement de jeux massivement multi-joueurs.

Il manque clairement quelque chose dans les outils actuels pour les rendre utiles à la conception de jeux innovants du point de vue des interactions.

4.2.5 Insuffisance des méthodes de développement actuelles

Le processus historique utilisé dans l'industrie pour le développement de jeux-vidéo ne convient plus pour la conception d'un jeu massivement multijoueurs. En effet, l'époque où un bon jeu vidéo se faisait à trois dans un garage, avec un développeur, un graphiste et un *game-designer*, est définitivement révolue avec ces projets impliquant parfois une centaine de personnes et prenant plusieurs années de développement.

Les politiques de gestion de projet sont donc en train de changer. Alors qu'historiquement le développement de jeux-vidéo est longtemps resté un secteur artisanal comparé aux autres développements logiciels, cette industrie se trouve désormais confrontée à la réalisation d'applications parmi les plus contraignantes. Nous avons vu dans le chapitre 3 comment l'industrie du jeu-vidéo tente de s'adapter, et nous avons étudié et critiqué les préconisations

des experts pour la réalisation d'un jeu massivement multi-joueurs.

Nous pensons donc que les méthodes utilisées actuellement dans l'industrie ne conviennent pas, et que les méthodes agiles comme *eXtreme Programming* [35, 7] présentées dans la partie précédente sont une alternative idéale pour la réalisation d'une application aussi complexe.

4.3 Notre proposition : un outil et une méthode

Le prototypage est un bon moyen de démontrer la faisabilité d'une nouvelle technologie dans l'industrie informatique en général, et nous avons vu dans le chapitre 3 comment les méthodes basées sur la réalisation de prototypes successifs permettent de sécuriser le déroulement d'un projet.

Les avantages du prototypage sont aussi valables pour l'industrie du jeu vidéo [73]. Dans le cadre du développement d'un jeu massivement multi-joueurs, la conception des interactions est critique, et directement liée au chiffrage financier des solutions de déploiement et d'hébergement effectué au démarrage du projet, surtout en ce qui concerne la somme des ressources à déployer (nombre de machines serveur, bande-passante). Un prototype a le bénéfice supplémentaire d'encourager l'innovation en fournissant les moyens de détecter les interactions critiques et d'expérimenter différentes recettes techniques avant l'étape du beta test.

Cependant, s'il est développé à partir de rien, un prototype demande en effet, comme le soulèvent les détracteurs de cette solution, un investissement considérable en temps de développement, même s'il n'est pas entièrement fonctionnel.

Il y a deux manières de résoudre ce dilemne :

- utiliser un cycle de vie du développement basé sur le raffinement de prototype, comme les méthodes agiles que nous avons décrites dans le chapitre précédent qui commencent à devenir assez populaires pour que certaines sociétés de jeux-vidéo commencent à s'y intéresser [29] au moins dans le cadre des jeux traditionnels ;
- disposer d'un outil de prototypage qui facilite l'établissement de la correspondance entre les interactions voulues fonctionnellement et les solutions techniques mises en œuvre, et qui soit capable de simuler des

conditions de déploiement réalistes pour valider cette correspondance.

Dans le premier cas, le code du prototype ne sera pas perdu, car raffiné, et réutilisé tout le long du cycle de développement. Les méthodes agiles ont d'autres avantages les rendant particulièrement bien adaptées au domaine du jeu-vidéo, par rapport aux autres méthodes basées sur le raffinement de prototypes successifs :

- elles sont adaptables à des projets de grande taille ;
- elles sont centrées sur les qualités des développeurs, généralement très compétents dans cette industrie attractive pour un professionnel en informatique ;
- elles permettent d'obtenir un logiciel plus facilement maintenable, quand la durée de vie d'un jeu massivement multi-joueurs se compte en années.
- leur principal inconvénient, qui est l'implication totale du client dans le projet, est réglé par le fait que le client sera ici le ou les *game-designers* du jeu massivement multi-joueurs.

Dans le deuxième cas, même si le prototype est jeté, on limite notablement l'investissement en temps nécessaire à sa réalisation.

Nous pensons que ces deux solutions doivent être rassemblées en une seule. En suivant l'exemple des outils qui existent pour réaliser des jeux-vidéo classiques, nous pensons qu'il est possible de proposer un modèle de développement pour les serveurs et les clients d'un jeu massivement multi-joueurs qui ne souffre pas du manque de généricité des solutions existant actuellement. Une approche basée sur des modules élémentaires, couplée avec des outils d'analyse et de prototypage, permet de détecter les fonctionnalités critiques. La modularité d'une telle approche aide à améliorer le prototype en prenant en considération les résultats d'analyses effectuées tout le long du cycle de développement, bien avant le début de la phase d'*alpha-test*, réduisant ainsi le risque de découvrir des défauts majeurs de conception trop tard pour le succès du projet.

4.4 Méthodologie de spécification du *framework*

Le travail réalisé dans le cadre de cette thèse consiste en la conception d'un *framework* (voir glossaire) sur lequel pourra s'appuyer l'outil évoqué dans la section précédente.

Nous allons exposer dans cette partie le cheminement qui nous a amené à la conception de ce *framework*, qui a été décrit tout d'abord dans [15], puis, de manière plus approfondie et détaillée dans [16].

Ce *framework* a donc pour but de servir de base à un environnement de développement pour encadrer la réalisation de jeux massivement multi-joueurs. L'outil final devra permettre de prototyper et de tester rapidement l'adéquation des interactions fonctionnelles exigées par le *game-play* et les solutions techniques utilisées pour ce faire, et d'accompagner le reste du développement dans le cadre de l'utilisation d'un processus agile. Cela implique certaines propriétés que le *framework* devra vérifier.

4.4.1 Philosophie générale

Il y a deux principaux écueils à éviter lors de la conception de ce *framework*.

Tout d'abord, la plupart des *frameworks* existants sont de trop haut niveau et manquent de souplesse en ne permettant pas d'adapter leurs composants. Comme nous l'avons vu précédemment, il n'est pas possible de concevoir un *game-play* novateur lorsque les solutions techniques sont figées. Il est donc probable que la solution que nous cherchons soit plus bas-niveau que celles qui existent actuellement, et nous nous interdirons de fixer les choix techniques.

Cependant, une solution trop bas-niveau risque de ne rien résoudre du tout en restant au niveau d'un langage évolué. Plus la solution sera bas niveau, plus le temps de développement en utilisant le *framework* se rapprochera d'un temps de développement en partant de rien.

Le compromis que nous avons adopté consiste à donner à l'utilisateur

une vision bas niveau du *framework*, pour lui laisser le choix de la solution à chaque problème donné, mais de faciliter au maximum la mise au point de cette solution. Le *framework* ne doit pas résoudre les problèmes, mais simplifier leur expression et leur traitement.

Pourquoi, alors, ne pas avoir choisi de définir plutôt un langage dédié ? En fait, le niveau de détail obtenu sera bien celui d'un langage évolué, dédié à la réalisation d'applications distribuées. Mais un des problèmes auquel on se frotte lors de la conception d'un langage est celui des primitives. Or, notre *framework* doit être ouvert, et permettre de manière naturelle l'adjonction de nouvelles primitives. La distinction langage haut niveau contre *framework* est donc principalement liée à un choix actuel de facilité d'implémentation pour des besoins d'ouverture à de nouveaux composants primitifs. Mais on ne s'interdit pas de réaliser plus tard le noyau d'un langage correspondant à la sémantique de la partie du *framework* représentant son cœur, son modèle de calcul, cette partie n'étant pas conçue pour être étendue par l'utilisateur.

4.4.2 Focalisation sur la mise au point des interactions

La différence technique majeure entre un jeu multi-joueurs en ligne et un jeu classique est bien l'interaction qui existe entre les joueurs et le reste du monde virtuel situé sur des machines distantes.

De même, la différence majeure entre un jeu multi-joueur et une application distribuée quelconque est la variété des interactions et la difficulté de leur calibrage.

C'est pourquoi nous nous sommes concentrés sur l'aide à la mise au point de ces interactions. Le travail présenté dans cette thèse ne concerne pas les aspects communs aux autres jeux-vidéo comme par exemple, les problématiques d'imagerie numérique, ou d'intelligence artificielle pour les personnages non-joueurs. Elle ne concerne pas non plus les problématiques de persistance et d'ingénierie de base de données communes aux applications distribuées en ligne plus classiques.

Cependant, pour pouvoir finaliser un projet de jeu massivement multi-joueurs, ces aspects doivent être pris en compte dans l'ébauche de notre solution. Le travail que nous avons réalisé se veut donc ouvert afin de pouvoir intégrer ultérieurement les autres technologies nécessaires.

4.4.3 Factorisation des aspects communs à la famille d'application visée

Concevoir un *framework* dédié à une certaine famille d'applications est un exercice délicat. Ce travail commence en général par une phase d'étude d'un certain nombre d'applications correspondant à ce qu'on veut pouvoir réaliser. Cependant, si la portée des applications étudiées n'est pas assez complet, le *framework* ne sera pas générique, (ce qui risque d'être le cas quand on veut l'utiliser pour développer des applications innovantes de toutes façons). D'un autre côté, il est pratiquement impossible d'extraire les points communs d'un trop grand nombre d'applications.

Au lieu d'étudier les jeux eux-même, nous avons étudié l'état de l'art des techniques actuellement utilisées pour la réalisation de divers types d'applications de la famille «Monde Virtuel», comme préconisé dans [38]. C'est le travail qui a été présenté dans le second chapitre de ce manuscrit.

Nous avons étudié précisément quels aspects de ces applications pouvaient modifier l'utilisation des ressources rentrant en ligne de compte dans le calibrage des interactions. Nous avons montré dans quelle mesure chaque solution visant à améliorer la qualité d'une caractéristique technique avait un impact sur les autres.

Cela nous a fourni les clés pour détecter les aspects devant rester ouverts et flexibles afin de permettre un éventail maximal de choix pour la réalisation des caractéristiques de l'application, et ceux pouvant être factorisés à l'intérieur du *framework*.

4.4.4 Détection du niveau d'abstraction et du degré de souplesse du *framework*

L'état de l'art présenté dans le deuxième chapitre de ce manuscrit a eu une conséquence majeure sur le niveau d'abstraction du *framework* : il est impossible de fournir une architecture de communication globale. L'impact des choix d'architecture sur l'utilisation des ressources réseau est trop importante pour pouvoir prétendre à la généricité.

Une autre conséquence majeure est que le modèle de communication de

chaque événement du jeu ne peut lui même être figé. Le calibrage du protocole de mise à jour de chaque donnée de l'application est là encore intrinsèquement lié à la consommation des ressources réseau. Compte tenu des interactions fonctionnelles à mettre en place, un calibrage très fin et spécifique à l'application peut être nécessaire.

C'est pourquoi le *framework* que nous proposons est basé sur les données répliquées sur les différents hôtes de la distribution. L'utilisateur doit pouvoir décider quelles données sont répliquées sur quels hôtes de la distribution, compte tenu de l'architecture qu'il a lui même choisi. Ainsi, il peut décider d'une architecture serveur divisée en service, d'une architecture centralisée, ou même d'une architecture *peer to peer*. De plus, le modèle de communication ainsi défini pour chaque donnée n'est pas nécessairement global. Chaque donnée peut avoir son propre modèle.

Enfin, la manière dont les données sont répliquées, c'est-à-dire la façon dont les données sont traitées par la couche réseau, est entièrement ouverte. Quelques briques de base paramétrables sont proposées en bibliothèque additionnelle au *framework*, et nous laissons la possibilité d'augmenter à volonté cette bibliothèque.

4.4.5 Pré-requis pour une intégration dans un environnement de développement

Enfin, puisque nous voulons ultérieurement intégrer le *framework* dans un environnement de développement, quelques autres aspects sont à prendre en considération, dont certains de nature purement ergonomique.

4.4.5.1 Différents niveaux de granularité

Nous avons vu dans les paragraphes précédents que nous avons finalement choisi un modèle de *framework* très bas niveau, où l'utilisateur doit lui même définir de quelle manière les données répliquées qui composent l'application seront traitées.

Ainsi, le découplage entre les données (organisées en état) et leur comportement différencie notre approche des solutions orientées objet, dont le

paradigme repose justement sur l'association des données et des traitements, de même que des approches orientées agent définies dans [87] ou [21].

Or, ce niveau de détail est trop fin pour pouvoir être utilisé à terme dans un outil de prototypage. Nous avons donc défini un modèle d'organisation des données en états. L'utilisateur peut ainsi factoriser les données en ensembles de valeurs à traiter de manière similaire.

Les états eux mêmes peuvent se composer arbitrairement de manière à ce qu'intuitivement, certains états définis par l'utilisateur puissent correspondre aux objets du jeu. Ainsi, une fois les données correctement organisées en états, l'utilisateur peut réfléchir et modéliser les interactions de manière naturelle et relativement intuitive pour un non programmeur.

L'avantage de cette solution est qu'une fois qu'un utilisateur averti a effectué la première étape de conception, un utilisateur moins averti, de profil intégrateur ou *game-designer*, peut lui-même jouer avec le paramétrage de l'application pour tester la qualité des interactions.

En fait, notre solution est à granularité arbitraire. Il sera toujours possible, en cas de besoin, de définir une interaction liée à une donnée de manière très fine et spécialisée par rapport aux autres données de l'application, mais en jouant avec la composition des états, on pourra toujours obtenir un niveau d'abstraction plus élevé.

Notre modèle permet ainsi, par une conception adéquate de l'utilisateur, de mettre en œuvre des approches orientées objet ou agent pour certaines données de l'application. Mais sans pour autant imposer que la totalité des données doive suivre l'un ou l'autre de ces paradigmes.

Cette possibilité de modélisation en plusieurs temps permet également de pallier le principal inconvénient du choix d'un *framework* bas niveau : les jeux simples resteront simples à prototyper, une fois les bonnes briques de base disponibles.

4.4.5.2 Un boîte à outils extensible et paramétrable

Nous avons vu dans les paragraphes précédents que le *framework* doit rester ouvert à des extensions concernant la manière dont sont traitées les fonctionnalités, notamment la façon dont sont propagées les données, ou celle

de calculer à quels hôtes les envoyer.

Nous obtenons ainsi un découplage entre les fonctionnalités et leur traitement, qui peut se rapprocher de l'objectif recherché par les techniques de meta-programmation comme le modèle de programmation par *aspects* [19], grâce à un *framework* ouvert qui permet à l'utilisateur de connaître, de modifier ou d'étendre les différentes implémentations d'une même fonctionnalité. Pour montrer l'adéquation de notre solution, nous proposons une boîte à outils minimale et paramétrable permettant de mettre en application quelques recettes techniques vues dans l'état de l'art présenté dans le deuxième chapitre, et pouvant ainsi servir de modèle à la réalisation ultérieure d'autres briques de base.

4.4.5.3 Du déterminisme et une sémantique bien définie

Enfin, un bon outil de développement doit pouvoir fournir des outils d'analyse du code produit, statiques et dynamiques.

De plus, la mise au point d'une application est toujours facilitée lorsque l'on peut facilement en tracer et en reproduire l'exécution. Or dans le cas d'une application en ligne, cette tâche est compliquée par son caractère distribué : il est très difficile de reproduire à l'identique le fonctionnement de la totalité de l'application, quand les protocoles Internet utilisés ne fournissent pas de garantie d'ordonnancement des messages ou d'assurance qu'ils seront bien reçus. Si on ajoute à cela le fait que beaucoup d'algorithmes utilisés dans le développement des jeux massivement multi-joueurs sont plus naturellement décrits en utilisant un paradigme de programmation concurrente (un système de processus légers par exemple), on obtient des applications très difficiles à corriger et à maintenir.

C'est pourquoi nous avons choisi d'introduire une forme légère de concurrence dans le *framework* lui-même. Le modèle que nous avons conçu est inspiré du modèle *Fair-Thread* (voir section 3.1.3, page 75) développé par Frédéric Boussinot [20]. Ce modèle est déterministe et sa définition sémantique est claire, ce qui facilite son utilisation et sa mise au point. L'adaptation que nous en avons faite rend notre modèle de description des interactions et de comportements des états du jeu facile à exprimer dans un contexte de programmation concurrente.

Une application développée à l'aide de notre *framework* n'est bien sûr pas reproductible dans l'absolu, puisqu'il s'agit d'une application distribuée sujette aux aléas des communications sur l'Internet. Mais il permet à l'utilisateur de disposer d'un cadre clair et bien défini, qu'il peut utiliser pour circonscrire les problèmes rencontrés au cours du développement en sous ensemble pour lequel le fonctionnement de l'application est reproductible (par exemple en scénarisant les événements provenant des autres hôtes de la distribution). Et cela tout en gardant la puissance d'expression que fournit le modèle de programmation concurrente.

4.5 Modèles de réplication et réflexes d'états du jeu

Dans cette partie, nous allons donner une première présentation informelle du modèle réalisé au cours de cette thèse.

4.5.1 Un modèle d'interactions basé sur la définition d'états

Un jeu massivement multi-joueurs peut être vu comme un ensemble de données répliquées sur tous les hôtes de la distribution. Ces données servent à représenter les objets du jeu et nous avons déjà vu que le principal problème dans la mise au point de ces applications était la manière dont les données doivent être répliquées.

Un des aspects à prendre en compte dans la réplication des données est l'architecture distribuée utilisée : par exemple, dans une architecture logique clients-serveur à base de *proxies*, où un serveur central est responsable de gérer la cohérence de l'état de l'application et où les *proxies* sont responsables de la gestion et de l'optimisation des communications en provenance des clients, l'état du jeu est répliqué sur le serveur central et sur les clients de l'application. Dans une architecture basée sur la division en zones géographiques du monde virtuel, seulement une partie de l'état du jeu doit être répliquée sur chaque serveur de zone, et sur les clients qui sont actuellement connectés sur ce serveur de zone.

Pour donner un exemple concret, la position de joueurs dans la région du monde virtuel gérée par un serveur de zone peut être uniquement gérée par ce serveur.

Comme nous l'avons vu dans le chapitre 2, l'architecture de l'application et le choix de l'ensemble des hôtes de l'application pour lesquels les données doivent être répliquées sont des problèmes de conception, qui ont un impact sur la qualité de l'interaction à laquelle ils sont liés.

De plus, il serait intéressant d'utiliser un modèle de communication différent selon la nature des données à répliquer : la gestion de certaines données pourrait être alors centralisée sur un seul serveur logique quand la consistance de l'état de l'application est un facteur très important, tandis que la réplication des données intervenant dans des interactions pour lesquelles l'aspect temps réel prime pourrait suivre un modèle de communication exploitant des responsabilités plus distribuées des serveurs. Le modèle que nous proposons ne se base donc pas sur une architecture de communication pré-définie.

Dans le modèle que nous définissons, l'entité de base que nous manipulons est un *état*.

Définition 4.5.1 *Un* **état** *encapsule un ensemble de données corrélées : elles ont les mêmes propriétés de réplication.*

Les données ne peuvent pas être partagées par deux états différents.

Un état peut avoir des sous-états. Les données représentant l'application sont donc strictement organisées en arbre.

Intuitivement, un état peut servir à représenter un objet du jeu.

4.5.2 Modèles de réplication

Quand un joueur interagit avec un objet du jeu sur son propre terminal, il se produit une modification de l'état du jeu sur sa machine cliente. Cette mise à jour de l'état du jeu doit être propagée au reste de l'application (même si ce changement d'état peut parfois seulement être l'enregistrement de la dernière commande provoquée par le joueur avant l'envoi sur le réseau). De la même manière, le résultat d'un algorithme comme la description du comportement

d'un personnage non joueur, ou n'importe quel autre processus s'effectuant sur le serveur intervient sur l'application et provoque des changements de l'état du jeu. Ainsi, modéliser une interaction dans notre modèle consiste à définir quand, comment et où la mise à jour d'un état doit être propagée le long de l'architecture de distribution de l'application, et quelles sont les conséquences de cette mise à jour sur les autres états. Nous proposons pour ce faire la notion de *modèle de réplication*.

4.5.2.1 Terminologie des modèles de réplications

Un modèle de réplication représente donc le moyen utilisé par un hôte de l'application distribuée pour propager la valeur de l'état aux autres hôtes.

Un modèle de réplication comporte trois attributs, définis comme suit :

Définition 4.5.2 *La* **portée** *est la liste des hôtes à qui propager la mise à jour de l'*état.

Par exemple, si le changement d'état correspond à un déplacement d'un joueur ou d'un personnage non-joueur, seuls les joueurs qui peuvent effectivement observer ces modifications, de par leur localisation dans le monde virtuel, ont réellement besoin de connaître ces modifications.

Définition 4.5.3 *La* **temporalité** *décrit quand propager la mise à jour de l'état.*

Cela peut être défini en donnant un rythme périodique auquel correspondront les réplications, ou en donnant une condition sur les valeurs d'un ou plusieurs états du jeu.

Définition 4.5.4 *Les* **propriétés de communication** *définissent la manière dont la mise à jour de l'état doit être propagée.*

Cet attribut est fortement lié au modèle de communication bas niveau, aux propriétés des protocoles utilisés et inclut par exemple les aspects de fiabilité, d'ordonnancement, de cryptage ou de compression.

Un modèle de réplication est donc dépendant du contexte de l'état du jeu : chaque attribut peut être défini par rapport à un ensemble fini de valeurs locales des états du jeu. Par exemple, si on considère un état représentant le déplacement d'un joueur dans le jeu, on peut vouloir utiliser une technique de filtrage pour décider que seuls les hôtes clients dont l'avatar est à une certaine distance de l'avatar du joueur qui se déplace ont besoin de recevoir la mise à jour. Dans ce cas, l'attribut **portée** consiste à passer en revue les positions des autres joueurs de la zone, afin de décider à quels clients l'état doit être propagé.

Il est possible de combiner plusieurs modèles de réplication pour la gestion des mises à jour d'un seul état : certaines données peuvent en effet nécessiter de répondre à différentes exigences de réplication.

Par exemple, considérons un état représentant la position actuelle de l'avatar d'un joueur donné sur un serveur de zone dans le cadre d'une architecture de distribution hiérarchique où un serveur central gère plusieurs serveurs de zone, chacun dédié à la maintenance d'un sous-ensemble de l'état du jeu suivant les régions du monde virtuel. Plusieurs modèles de réplication définissent la manière plus ou moins temps réel dont la position de l'avatar concerné doit être propagée aux autres clients de l'application, en fonction de sa distance par rapport aux positions des avatars des joueurs correspondants. Le serveur central, relié à une base de données assurant la persistance de l'application, peut utiliser un modèle de réplication pour être prévenu périodiquement afin d'enregistrer de temps en temps la position du joueur, en cas de défaillance de l'application. Il peut en utiliser un autre pour enregistrer la position de l'avatar en cas de déconnexion du joueur.

4.5.2.2 Réflexes

Afin de pouvoir définir les conséquences d'un changement d'état sur un hôte de l'application, on définit également la notion de **réflexe**.

Définition 4.5.5 *Un* **réflexe** *est une partie de code défini par l'utilisateur, opérant sur un ensemble d'états de l'application, et lié à l'attribut* temporalité *du modèle de réplication. Lorsque la* temporalité *est vérifiée, la réplication est déclenchée en utilisant les autres attributs du modèle de réplication, puis le réflexe est exécuté.*

Un réflexe peut servir à effectuer des calculs arbitrairement complexes, allant de la simple réinitialisation de la valeur d'un état à la description d'un comportement autonome d'un objet du monde virtuel.

Les réflexes peuvent également être utilisés pour définir l'impact d'un changement d'état sans qu'il soit nécessaire que cet état soit sujet à une réplication. Dans ce cas, le réflexe est défini sans liaison avec un modèle de réplication, mais avec un attribut temporalité qui permet de spécifier son déclenchement.

Les réflexes, une fois déclenchés par la vérification de leur temporalité, sont exécutés par le *framework* selon une sémantique que nous étudierons dans le chapitre suivant.

Nous pouvons désormais donner une définition d'un modèle de réplication d'un état :

Définition 4.5.6 *Un modèle de réplication d'un état (définition 4.5.1) est la donnée des trois attributs portée (définition 4.5.2), temporalité (définition 4.5.3) et propriétés de communication (définition 4.5.4), et d'un éventuel réflexe post-réplication (définition 4.5.5).*

4.5.2.3 Intuition sur un exemple simple

Nous reprenons le cas de figure évoqué précédemment des modèles de réplication de la position de l'avatar d'un joueur, dans le cadre d'une architecture clients-serveur centralisée où des serveurs de zone sont en charge de chaque zone géographique du monde virtuel.

Le tableau 4.1 décrit ce qui se passe sur un serveur de zone lorsqu'un joueur se déplace. L'état **position** représente la position de l'avatar du joueur, et on décrit les modèles de réplications qui lui sont associés :

- Le premier modèle de réplication (*Voisins*) utilise les états correspondant à la position des avatars connectés au même serveur de zone pour filtrer quels sont les avatars voisins étant donnée une certaine distance (qui peut elle-même également être définie comme un état) pour définir la portée. La temporalité est conditionnée par le changement d'état et la propagation de ce changement est définie comme non fiable et non ordonnée.

- Le deuxième modèle de réplication (*Zone*) a une portée qui est l'ensemble des clients connectés au même serveur de zone, avec une temporalité périodique de 500ms. Les propriétés de communication suivent un protocole non fiable et ordonné comme dans le modèle précédent.
- Le troisième modèle de réplication (*Central*) a une portée constituée uniquement du serveur central, avec une temporalité périodique de 20 secondes, et utilise un protocole fiable et ordonné. Elle permet de gérer la persistance de l'application.

Réplications	*Voisins*	*Zone*	*Central*
portée	ensemble des voisins, calculé à l'aide d'un filtre	clients de la zone	serveur central
temporalité	changement de l'état **position**	toutes les 500 ms	20s
propriétés de communication	non fiable, ordonné	non fiable, ordonné	fiable

FIG. 4.1 – Réplications du mouvement d'un avatar sur un serveur de zone

4.5.3 Vue d'ensemble

La figure 4.2 décrit l'interaction des différents processus intervenant dans la partie cliente d'une application réalisée à l'aide du *framework*. Ce fonctionnement est classique. Une mémoire partagée par les différents processus contient l'ensemble des états définis pour représenter l'état de l'application. Les événements provenant du réseau et les événements provoqués par le joueur à travers l'interface utilisateur du jeu provoquent des changements dans cette mémoire partagée. Le joueur perçoit l'état du jeu à travers un ou plusieurs observateurs utilisés par l'interface graphique, comme un moteur de rendu 3D par exemple. Un processus particulier gère le déclenchement des modèles de réplication et l'exécution des réflexes définis par l'utilisateur. Ce processus est à la fois observateur de l'état du jeu et producteur de modifications des états.

Les messages arrivant par le réseau doivent correspondre à la réplication d'un état par un hôte distant, et cet hôte doit être autorisé localement à modifier l'état pour que la mise à jour soit prise en compte. Si l'état n'existe

pas sur localement et que l'hôte distant y est autorisé, le message réseau de réplication provoquera la création de l'état local.

Des processus supplémentaires peuvent naturellement être intégrés à l'application pour travailler sur ou à partir de la mémoire partagée, mais nous verrons plus tard que cela peut avoir des conséquences sur la clarté de la sémantique de l'application, car ces processus ne s'exécuteront pas selon le calcul que nous avons défini pour l'exécution des réflexes.

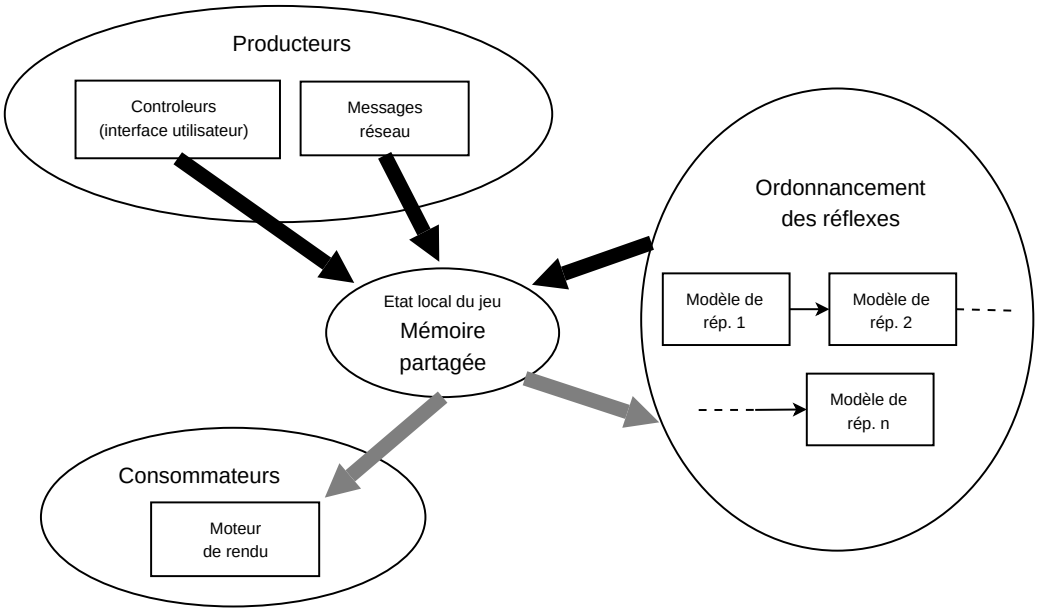


FIG. 4.2 – Architecture des processus de l'application sur un hôte

La figure 4.3 décrit l'organisation des différents éléments intervenant dans le fonctionnement de l'application du point de vue de l'utilisateur du *framework*. Pour chaque différent type d'hôte de l'application (les clients et les hôtes de l'architecture du serveur), l'utilisateur a pour tâche de définir les états à partir des données permettant de représenter l'application.

Une fois les données organisées en états, il utilise les composants de base du *framework* afin de définir les éventuels modèles de réplication pour ces états. Comme représenté sur la figure, un état n'a pas nécessairement de modèle de réplication : il n'est pas toujours nécessaire de propager la modification d'une donnée, soit parce qu'elle est privée à l'hôte et est utilisée pour

la valeur de ses données pour les calculs définis dans les réflexes ou la description des attributs, soit parce que celui-ci n'a pas la responsabilité de sa mise à jour qui s'effectue au travers du réseau par un hôte distant. Un même état peut avoir plusieurs modèles de réplications correspondant à différents besoins de persistance.

Le corps du *framework* s'occupe de déclencher les modèles de réplication et d'exécuter avec une sémantique bien définie les réflexes attachés ou non à ces modèles, modifiant éventuellement les différents états de la mémoire partagée par les processus locaux de l'application.

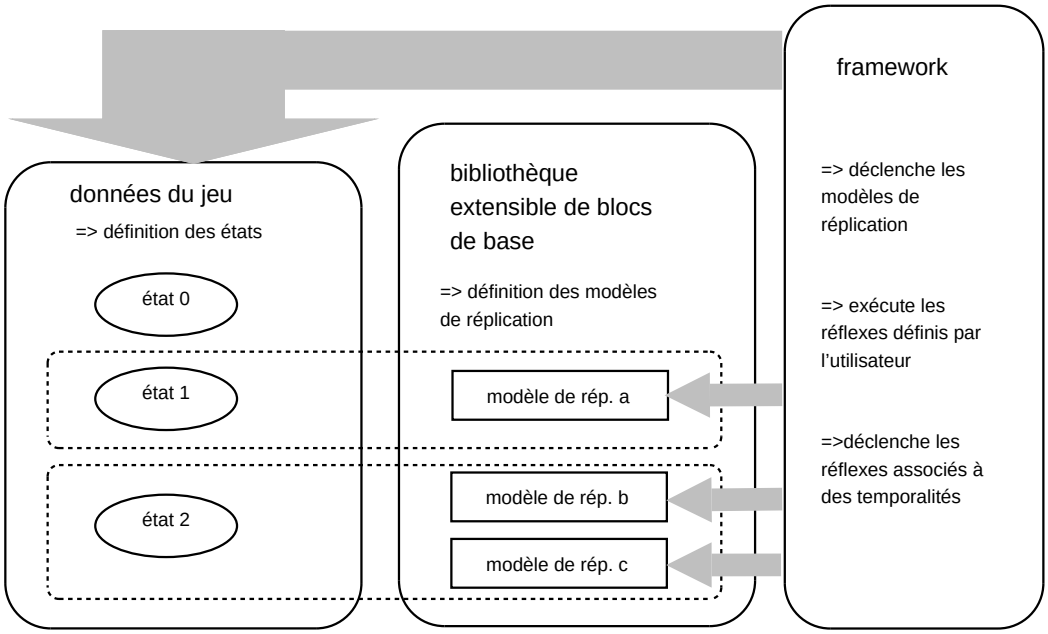


FIG. 4.3 – Modèles de réplications d'états sur un hôte de la distribution

4.5.4 La bibliothèque des blocs de base

Comme nous l'avons vu, l'utilisateur du *framework* que nous proposons s'appuie sur une bibliothèque extensible de blocs de base éventuellement paramétrables. Ceux-ci sont rangés en plusieurs grandes familles, et servent à construire les modèles de réplication ou les réflexes sans réplication attachés aux états.

- les *propriétés de communication* des réflexes.
- les *portées*, qui peuvent être des ensembles statiques de destinataires, ou calculés dynamiquement selon les valeurs d'états de l'application.
- des *réflexes* liés à une *temporalité*, décrivant des comportements d'états, paramétrables par ces derniers, qui peuvent ainsi permettre de constituer des bibliothèques de description de comportements d'objets du monde virtuel.
- des *modèles de réplication* génériques paramétrables, dont la temporalité (les conditions de déclenchement) est retrouvé fréquemment : par exemple une réplication instantanée lors d'un changement d'état.

Les deux dernières familles de composants se situent à un niveau d'abstraction supérieur, les modèles de réplication génériques étant paramétrables par des portées et des propriétés de communication.

L'utilisateur peut étendre cette bibliothèque en utilisant des interfaces fournies par le *framework*, à la manière des *Hooks* décrits dans [38].

4.5.5 Exemples de modélisation, cas d'école

Nous présentons ici trois exemples simples, modélisés volontairement de manière très détaillée, pour donner au lecteur une intuition de la manière d'utiliser le modèle pour concevoir quelques interactions classiques intervenant dans un jeu.

4.5.5.1 Compétition pour accéder à un objet : ouverture d'un coffre

Dans le jeu, un coffre fermé par un cadenas peut être ouvert par les joueurs disposant du bon code. Deux joueurs essaient d'ouvrir le code simultanément.

L'architecture distribuée choisie par les concepteurs est une architecture clients-serveur classique. Le concepteur des interactions fonctionnelles décide que le contenu du coffre est accessible seulement au joueur qui l'ouvre le premier avec le bon code. Il décide également que l'attribution du verrou logique n'a pas à être équitable : si les deux joueurs entrent le bon code exactement au même moment, c'est celui dont l'information sera traitée en premier par le serveur qui pourra explorer le contenu du coffre. C'est donc le

joueur qui aura le moins de latence qui sera favorisé. Dans cet exemple, les objets du jeu concernés sont le *coffre*, et les *avatars* des deux joueurs.

La modélisation bas-niveau que nous donnons est destinée à expliquer comment traiter un modèle de communication par requête, même si celles-ci sont moins naturellement traitées que dans le cas d'objets répliqués. Bien sûr, il y a bien d'autres manières de modéliser cette interaction.

Parmi les états définissant le *coffre*, on trouve :
- **utilisateur :** contient l'utilisateur qui a obtenu le verrou sur le coffre, une valeur par défaut s'il n'y en a aucun.
- **code entré :** la valeur du dernier code entré pour ouvrir le coffre.
- **code secret :** le code secret permettant d'ouvrir le coffre.
- **demandeur :** le client demandant actuellement accès au coffre.

Le tableau 4.4 représente les modèles de réplication associés aux états définissant le coffre et les réflexes. Quand les joueurs entrent le code, l'état

états répliqués	***code entré***	***code entré***	***utilisateur***
localisation	client	serveur	serveur
portée	serveur		**demandeur**
temporalité	**code entré** modifié, non défaut	**code entré** modifié	**utilisateur** modifié
communications	fiable		fiable
états modifiés par les réflexes	**code entré**	**demandeur**, **utilisateur**	

FIG. 4.4 – Réplication et réflexes pour les états représentant le coffre

code entré est modifié sur chaque client. Les modèles de réplication côté client provoquent une communication immédiate de ce changement au serveur, puis le réflexe associé réinitialise la valeur de ces états clients à la valeur par défaut.

Côté serveur, lorsque la première communication de la nouvelle valeur de l'état **code entré** arrive, l'état **code entré** local est mis à jour. Le réflexe serveur associé au changement d'état met à jour l'état **demandeur** avec la valeur du client qui a effectué la mise à jour de **code entré** et fait quelques vérifications afin de modifier l'état **utilisateur** : si l'état **code entré** contient la même valeur que l'état **code secret** et si l'état **utilisateur** contient la valeur par défaut (pas d'utilisateur du coffre), il modifie dans l'état **utilisateur**

la donnée représentant le propriétaire du coffre pour l'attribuer au client demandeur. Sinon, il provoque également un événement de modification de cet état, mais sans modifier la donnée encapsulée représentant le propriétaire du coffre.

Puis, le **framework** applique le modèle de réplication pour l'état **utilisateur** : la valeur courante de l'état **utilisateur** est envoyée au client qui a demandé le verrou en utilisant la valeur de l'état **demandeur**.

Pour modéliser cet exemple, une fois qu'il a correctement défini les états, l'utilisateur du **framework** doit simplement décrire les attributs des modèles de réplication, et définir le code des réflexes. Ce travail consiste à décrire des traitements sur les états définis, et à utiliser des composants de base faisant partie de la bibliothèque.

4.5.5.2 Combat en temps réel dans une zone peuplée par des joueurs

On s'intéresse maintenant à l'interaction suivante : dans un jeu persistant d'action temps-réel de style FPS, un joueur tire sur un autre à l'aide d'une arme.

L'architecture du serveur de l'application suit un découpage de la géographie du monde virtuel en serveurs de zone, et un serveur central est en charge de la coordination des serveurs de zone et de la gestion des données persistantes de l'application. Chaque client est connecté au serveur en charge de la zone géographique du monde virtuel dans laquelle se trouve son avatar. Des techniques d'extrapolation sont utilisées côté client, pour anticiper les futures positions des avatars selon les valeurs précédentes de leurs directions et vitesses.

Si le joueur visé est frappé aux jambes, son total de points de vie et sa vitesse de course décroissent. Comme cette action peut se produire dans une région vaste et très peuplée du monde virtuel et qu'informer tous les joueurs en temps réel consommerait trop de ressources, le concepteur des interactions fonctionnelles décide que les proches voisins du joueur visé doivent pouvoir constater le plus rapidement possible la baisse de la vitesse de course du joueur touché, tandis que les joueurs plus éloignés n'ont pas besoin de constater ce changement aussi instantanément.

En ce qui concerne le total des points de vie du joueur touché, le concepteur décide qu'il doit être communiqué à tous les clients du serveur de zone de manière non fiable, et avec une très haute priorité. La «mort» d'un avatar est un événement crucial, qui doit être communiqué de manière fiable par le serveur de zone au serveur central, qui est en charge des données persistantes de l'application, et à tous les clients de ce même serveur de zone.

Ici, notre but est de décrire comment des techniques de filtrage selon les intérêts peuvent être modélisées. Nous allons décrire cette interaction à partir du moment où le serveur a déjà calculé que l'avatar visé a été touché. L'objet à décrire est l'avatar du joueur. Il comporte notamment trois états qui sont respectivement :

- **déplacement** : les caractéristiques du dernier mouvement de l'avatar, c'est-à-dire, en plus de sa position, sa vitesse et sa direction instantanée ;
- **points de vie** : la quantité des points de vie que l'avatar peut encore perdre avant d'être considéré comme décédé. Comme l'avatar peut être soigné par un joueur partenaire dans le même temps où il est touché par un autre, et que les événements correspondants peuvent arriver dans le désordre par rapport au moment où il se sont produits, on décide d'appliquer une tolérance : un avatar peut brièvement posséder un nombre de points de vie négatif sans être encore considéré comme décédé.
- **vivant** : une valeur indiquant si l'avatar est encore vivant ou non. Cet état n'est pas redondant avec le précédent suite à la tolérance décrite.

La figure 4.5 décrit les modèles de réplication sur le serveur de zone où l'avatar se situe au moment de l'action. Le processus serveur décide que l'avatar est touché, et met à jour les états précédemment décrits selon le *game-design* défini. Ensuite, il exécute les modèles de réplication pour chaque état dont l'attribut temporalité est vérifié. Dans la description de cette interaction, outre la manière dont on décide si l'avatar a été touché, la seule partie spécifique, décrite par l'utilisateur du *framework*, est la manière de calculer la liste des hôtes devant recevoir les mises à jour d'états répliqués. Comme ces techniques de filtrage sont des techniques couramment utilisées, on peut envisager de fournir cette partie spécifique comme un bloc de base paramétrable faisant partie d'une bibliothèque accompagnant le *framework*. On peut tout simplement décider si oui ou non deux avatars sont voisins par rapport aux valeurs respectives de leurs états *position*.

L'algorithme d'extrapolation côté client peut également être un bloc de

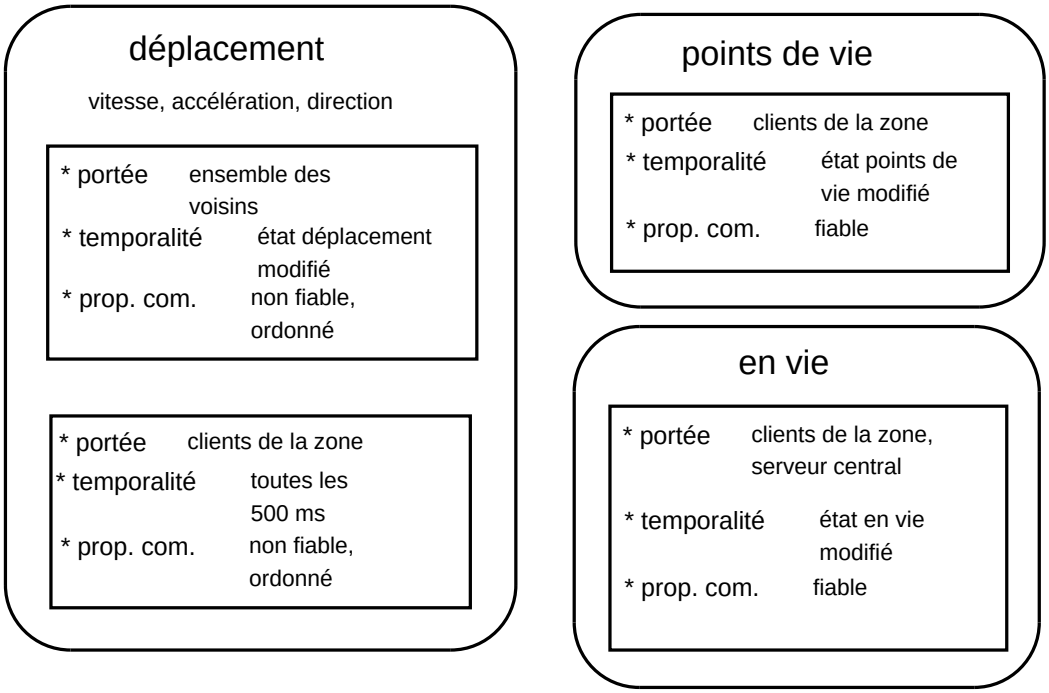


FIG. 4.5 – Réplications du mouvement d'un joueur sur un serveur de zone

base paramétrable inclu dans la bibliothèque intégrée au *framework*, et paramétrable par l'utilisateur.

En utilisant la modélisation de cette séquence à l'aide du *framework*, conjointement avec des outils de simulation réseau introduisant latence et perte de paquets, et avec un outil permettant de simuler la présence d'un grand nombre de clients, le *game-designer* peut alors tester si le *game-play* est réalisable et les choix techniques adéquats. Par exemple, il peut étudier l'impact de différents paramètres de filtrage sur la précision de l'algorithme d'extrapolation et tester si les caractéristiques de jouabilité restent satisfaisantes. Il peut aussi étudier l'impact d'une perte de paquets réaliste sur le nombre des points de vie par rapport à l'information de la mort d'un avatar, toujours afin de vérifier si les joueurs ne risquent pas d'être surpris par une évolution trop rapide de l'action, au cas où la plupart des paquets soient perdus.

4.5.5.3 Adaptation dynamique de consommation de bande passante

Notre modèle peut également être utilisé pour décrire un équilibrage dynamique de consommation de ressources, par exemple pour la consommation de bande passante.

Dans le cadre d'un jeu grand-public, conçu pour pouvoir être utilisé par des joueurs disposant de ressources hétérogènes en terme de bande passante, et qui n'observent pas les mêmes latences moyennes, le concepteur peut décider d'utiliser un bloc de base comportemental qui analyse dynamiquement un état représentant les capacités moyennes observées pour chaque client.

Ainsi, à l'aide des mêmes techniques de filtrage que celles de l'exemple précédent, mais en utilisant non pas la position de l'avatar mais les valeurs des états représentant les capacités du client, le concepteur peut modéliser un mode dégradé de comportement de l'application pour les joueurs disposant de faibles connexions.

4.6 Autres approches génériques

Dans cette partie, nous allons situer notre approche avec deux autres solutions récentes, proposant des *frameworks* génériques dédiés à la réalisation de jeux massivement multi-joueurs.

4.6.1 *Virtual Environment System Object Model* et JADE

Dans [79], Manuel Oliveira propose une architecture de développement flexible pour la réalisation de mondes virtuels allant des jeux massivement multi-joueurs aux FPS, constituée de quatre couches de développement orienté objet construites l'une au dessus de l'autre, nommée *Virtual Environment System Layered Object Model* (VESLOM). Ce modèle a été défini afin de résoudre le manque de souplesse des solutions existantes et de proposer une solution modulaire et évolutive, au fur et à mesure que des nouveaux besoins apparaissent. La volonté de fournir une approche générique est commune avec le travail réalisé dans cette thèse.

Voici les quatres couches successives sur lesquelles reposent VESLOM.

1. La **plate-forme universelle** rassemble les fonctionnalités nécessaires à tout développement de monde virtuel. Lorsqu'un système construit à l'aide de VESLOM devra évoluer pour s'adapter à de nouveaux systèmes et de nouvelles technologies, les modifications devraient donc être principalement localisées sur cette plate-forme. JADE (Java Adaptative Dynamic Environment), la proposition de plate-forme universelle développée par l'auteur en Java consiste en quatre modules. Le premier sert à identifier toutes les ressources de l'application, des textures graphiques aux éléments de code modulaire, qui sont fournies par les couches supérieures. Le deuxième module fournit un modèle événementiel souscription/publication pour la communication entre les composants du système. Un troisième module est dédié à la localisation de ressources, en local ou de manière distante. Le dernier module permet d'adapter des interpréteurs pour différents langages de description de contenu (par exemple *XML*), afin de découpler le code de l'application de son contenu et donc de permettre la réalisation de mondes virtuels de contenus différents à l'aide du même système.
2. Le rôle de la **couche réseau** est d'utiliser la plate-forme universelle pour implémenter des protocoles réseau plus élaborés pour la communication dans le monde virtuel.
3. La **couche logicielle** regroupe tous les composants nécessaires à la constitution d'un monde virtuel. Le but de cette couche est de construire un ensemble modulaire de composants. Cependant, il n'y a pas d'impératifs quant au nombre ou à l'autonomie de ces composants : cette couche est prévue pour être extensible. Le développeur peut ajouter de nouveaux composants, et définir des ensembles de composants dépendants les uns des autres.
4. La **couche applicative** regroupe tous les composants spécifiques au monde virtuel en cours de développement.

Cette approche est donc moins monolithique que celles que nous avons rencontrées en passant en revue les différentes solutions aux problèmes techniques rencontrés dans la deuxième partie de ce manuscrit.

Un des points communs avec le travail présenté dans cette thèse est l'utilisation d'une boîte à outils extensible et paramétrable, construite autour d'un modèle d'exécution (ici la plate-forme universelle) qui se veut générique. Le

modèle de construction en *oignon* de VESLOM nous paraît néanmoins moins souple et moins fin que celui que nous définissons, car les différentes couches sont construites les unes au dessus des autres. Dans notre modèle, les grandes familles de briques de base sont placées au même niveau et se combinent entre elles pour former la manière dont l'application se comportera.

De plus, la méthode accompagnant le modèle VESLOM reste guidée par l'architecture de conception du logiciel pour un hôte de l'application distribuée, alors que notre démarche est guidée par la définition des interactions le long d'une architecture de distribution. Une des conséquences de cette différence est que certaines fonctionnalités de la plate-forme universelle, comme les différents moyens d'accéder à une ressource distante (par HTTP ou Jini [4] dans JADE), deviendraient des briques de base dans notre approche.

4.6.2 OpenPING

Le système OpenPING [78] est une version améliorée du projet européen PING [83] dédié à la création d'une solution pour les applications de simulation massivement multi-utilisateurs tels que les jeux massivement multi-joueurs. Les produits du projet PING sont notamment la plate-forme Continuum [97], accompagnée d'un langage de description de comportement d'agents évoluant dans des mondes virtuels [21] (utilisant les principes de la programmation réactive, et déjà cité dans la partie de ce mémoire consacré à ce modèle de programmation), dont OpenPING reprend les principaux aspects, en y ajoutant des propriétés de réflexion.

Tout comme notre proposition, OpenPING est un *framework* orienté objet ouvert, qui donne une visibilité à l'utilisateur sur les cinq principaux services permettant de réaliser les fonctionnalités de l'application, sollicités lorsqu'un événement se produit dans l'application.

- Le service de *réplication* fournit des mécanismes de mise à jour périodique des données de l'application, pour des rythmes rapides, moyens ou lents.
- Le service de *concurrence* assure le transfert de propriété des objets du jeu entre les différents hôtes de l'application.
- Le service de *consistance* comprend des algorithmes de synchronisation des différents hôtes de la distribution, quand le service de réplication n'est pas utilisé ou pas suffisant. Il définit donc d'autres modèles de

communication.
- Le service de *gestion des intérêts* fournit des algorithmes de communications de groupe, basés sur les coordonnées spatiales des objets de l'application, ou sur un modèle publication/souscription.
- Le service de *persistance* est en charge de la maintenance de l'état de l'application, que ce soit dans la mémoire locale (pour des jeux par session par exemple) ou sur support externe comme une base de données.

La description des comportement des objets de l'application, qu'ils correspondent à des objets du monde virtuel ou à des mécanismes système, est faite dans une couche applicative de niveau d'abstraction supérieur.

Les quatres premiers services utilisent un composant en charge de la transmission des événements aux autres hôtes de l'application, consistant en un mode non fiable non ordonné (UDP), un mode fiable et paramétrable au niveau du composant, et un mode paramétrable au niveau de la couche applicative.

L'amélioration principale apportée au projet PING par Paul Okanda et Gordon Blair est l'ajout de propriétés de *réflexion* au système originel. Cette propriété est généralement définie comme la possibilité pour un système de raisonner sur sa propre structure et sa propre exécution à l'aide d'une représentation de lui-même. Le travail présenté dans [66] passe en revue quelques langages permettant d'exprimer cette propriété et décrit un exemple d'architecture réflexive pour un langage orienté objet. Les auteurs re-définissent cette propriété dans le cas particulier des mondes virtuels : *un principe de conception qui permet à une plate-forme de jeu et/ou à l'application d'avoir une représentation d'elle-même afin de rendre possible son adaptation à un environnement qui évolue.*

Les auteurs d'OpenPING utilisent donc ce concept afin de fournir un mécanisme qui permet à l'application de s'adapter automatiquement et dynamiquement à son propre état : par exemple, lorsque les communications se déroulent parfaitement bien au sein de l'application distribuée, la gestion des intérêts utilise le mécanisme reposant sur la localisation spatiale des objets du jeu, et s'adapte en passant à un modèle publication-souscription en mode dégradé, si le système observe que le réseau sature.

Il nous semble que ce système ouvert, d'une architecture plus complexe mais plus haut-niveau que la nôtre, est équivalent en terme d'applications

réalisables à notre modèle.

Le découplage entre les fonctionnalités à réaliser et la manière dont elles sont réalisées, au moyen de services extensibles, se rapproche d'un modèle de programmation par aspect [19]. Dans notre *framework*, la possibilité de définir plusieurs manières dont un état sera répliqué à travers plusieurs modèles de réplications, se rapproche d'une telle approche transverse. De plus, nous avons montré dans la section 4.5.5.3 (page 125) comment le découplage entre les états et les différentes manières de les répliquer peut être utilisé à profit pour fournir une application qui s'adapte automatiquement à un changement de contexte d'exécution (en l'occurrence la quantité de bande passante du client).

4.7 Synthèse

Le modèle que nous proposons est donc centré sur une modélisation très fine de la manière dont les données du jeu, organisées en états, doivent être répliquées le long de l'architecture distribuée. La conception d'une application utilisant ce modèle est donc guidée par la définition des interactions.

À la différence des modèles orientés objets et orientés agents, il découple les données des traitements, et de leurs comportements. Mais il est possible de réaliser des applications correspondant à ces paradigmes en modélisant de manière adéquate l'agencement des états et leur association avec des modèles de réplications et des réflexes sans réplications associées.

Nous avons également étudié dans cette partie d'autres propositions de *framework* génériques qui semblent disposer des mêmes propriétés de modularité que la nôtre, mais guidées par l'architecture de l'application ou une définition des services à mettre en oeuvre.

À notre connaissance, notre démarche est la seule qui prend en compte les impératifs liés à la construction d'un environnement de développement et de prototypage destiné à la mise au point des interactions pour les jeux massivement multi-joueurs.

Chapitre 5

Modèle de Réplication des Interactions

5.1 Introduction

Ce chapitre est dédié à la présentation plus en détail du modèle sous-jacent au *framework* réalisé dans le cadre de cette thèse.

Dans la première partie, nous décrivons le modèle de concurrence utilisé sur chaque hôte pour le calcul du comportement de l'application, modélisé à l'aide des états du jeu, et des réflexes définis par l'utilisateur.

Nous étudions ensuite plus en détail la notion d'état et leur modèle de composition, et expliquons de quelle manière cette composition est gérée aussi bien dans les calculs effectués par les réflexes que dans le cadre d'une réplication.

Nous formalisons ensuite la sémantique de notre modèle, en donnant quelques règles d'inférence décrivant les modifications de l'application lors de l'utilisation des instructions utilisateur.

Enfin, nous illustrons l'approche en donnant quelques exemples de modélisations correspondant à des situations qu'on peut rencontrer fréquemment en utilisant le modèle.

5.2 Modèle de concurrence

La programmation d'applications de type mondes virtuels que sont les jeux-vidéo se fait bien plus naturellement en utilisant le paradigme de la programmation concurrente. En effet, il paraît tout naturel de décrire séparément le comportement des différents objets constituant le monde, au lieu de décrire séquentiellement à chaque instant la manière dont l'état global du monde apparaît.

Ainsi, dans notre modèle, l'exécution d'un réflexe qui concerne un état/objet du monde peut représenter un traitement arbitrairement lourd en terme de ressources.

La concurrence est donc une nécessité pour permettre à l'utilisateur de partager les ressources entre les réflexes selon leur complexité.

Nous allons dans cette partie discuter et présenter la manière dont nous avons choisi d'introduire la concurrence dans le traitement des réflexes associés aux modèles de réplication.

5.2.1 Un ordonnanceur équitable

Comme nous l'avons vu dans le chapitre 3, Les approches préemptives traditionnelles de la programmation parallèle souffrent de plusieurs inconvénients par rapport aux objectifs que nous nous sommes fixés, et principalement à celui qui consiste à essayer de simplifier la vie de l'utilisateur. Ce style de programmation est complexe, à cause de la nécessité de contrôler l'accès à des données partagées, et de la difficulté à déterminer quand auront lieu les changements de contexte.

De plus, chercher les erreurs dans un programme écrit à l'aide de ce modèle devient très vite un cauchemar à cause de la non reproductibilité qu'il implique souvent, d'une exécution du programme à l'autre.

Le modèle que nous avons choisi est inspiré par le modèle *Cooperative Fair-Thread* [20], coopératif et équitable, étudié au chapitre 3 (voir section 3.1.3, page 75).

Notre modèle est coopératif, et donc plus facile à manipuler pour l'utili-

sateur qui contrôle, à l'aide d'une instruction *cooperate* quand chaque tâche rend la main à l'ordonnanceur pour qu'il passe à une autre tâche en cours. Le modèle d'ordonnancement des tâches est reproductible et équitable, donnant la main à chaque tâche à tour de rôle.

Ainsi, modulo les événements provenant du réseau et des autres processus extérieurs au *framework* pouvant modifier l'état local du jeu (comme par exemple une interface graphique), la sémantique de l'exécution du programme est clairement définie, facilitant les tests, la mise au point et la maintenance de l'application.

On se retrouve donc avec un modèle de concurrence à deux niveaux :
- le premier est préemptif, et consiste à gérer l'accès à la mémoire partagée des différents processus principaux : réception des réplications arrivant par le réseau, des événements de l'interface utilisateur, rendu graphique, et de l'ordonnanceur équitable qui organise l'exécution des réflexes. Il correspond au découpage présenté au chapitre précédent dans la figure 4.2 ;
- le deuxième niveau est coopératif et équitable. C'est l'ordonnanceur équitable des réflexes. Chacun des réflexes représente une tâche, qui sera exécutée par l'ordonnanceur dans un processus léger. Les réflexes sont attachés à la file d'exécution de l'ordonnanceur dans l'ordre dans lequel ils ont été déclenchés par leur temporalité.

Un réflexe est défini par l'utilisateur comme une suite de sections critiques d'instructions, entre chaque invocation de l'instruction *cooperate* qui redonne la main à l'ordonnanceur. Lorsqu'un réflexe est déclenché, le code correspondant est attaché à la file d'exécution de l'ordonnanceur.

Les segments d'instructions sont ensuite exécutés de manière équitable : quand plusieurs réflexes sont en cours d'exécution en parallèle, l'ordonnanceur exécute tour à tour dans l'ordre dans lequel elles ont été déclenchées, les premières sections critiques encore à traiter de chacun d'entre eux.

Quand un réflexe est attaché à la file de l'ordonnanceur, sa première section critique sera traitée dès que les réflexes en cours d'exécution seront terminés ou auront rendu la main à l'ordonnanceur.

Le mécanisme d'ordonnancement est illustré par la figure 5.1 : elle présente l'ordonnancement de l'exécution de trois réflexes dans trois processus légers, divisés en plusieurs sections critiques à l'aide de l'instruction *cooperate*.

Les deux premiers réflexes ont été ajoutés simultanément à la file d'exécution. Le troisième a été attaché à l'ordonnanceur à un moment donné de l'exécution des deux premières sections critiques des deux premiers réflexes.

Nous utiliserons dans la suite de ce document le terme d'*ordonnançable* défini de la manière suivante :

Définition 5.2.1 *Un* **ordonnançable** *est soit un* modèle de réplication *ayant défini un* réflexe post-réplication*, soit un* réflexe lié à un attribut temporalité *définissant son déclenchement mais ne donnant pas lieu à une réplication. Un ordonnançable est donc l'association d'une condition de déclenchement et d'un réflexe exécuté par l'ordonnanceur lors de la vérification de cette condition.*

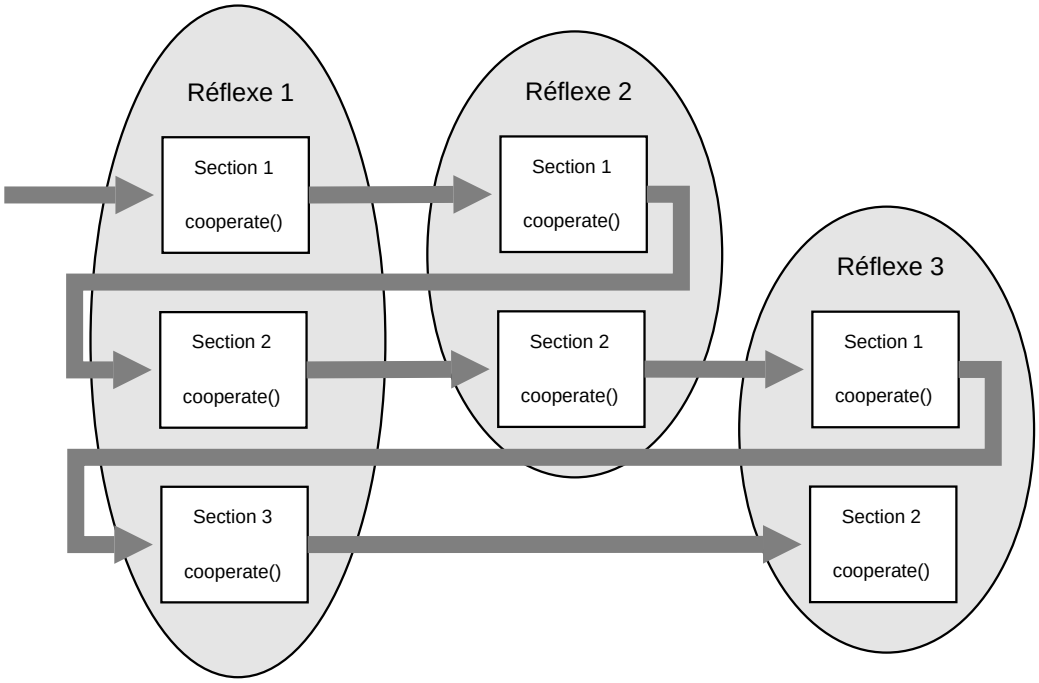


FIG. 5.1 – Exécution équitable des réflexes

5.2.2 Les copies d'états et les instructions *flash* et *update*

Il y a un autre problème à prendre en considération, introduit par l'utilisation d'un ordonnanceur pour gérer l'exécution des réflexes : certains calculs sont plus faciles à décrire par un réflexe utilisant les valeurs des états au moment de leur déclenchement : par exemple, le contrôle de la validité du mouvement d'un joueur, ou, plus généralement, quand la condition utilisée par l'attribut de temporalité de l'ordonnançable n'est plus vérifiée lors du traitement effectif du réflexe. Or, les valeurs des états intervenant dans le calcul peuvent avoir été modifiées avant l'exécution du réflexe par l'ordonnanceur, soit par un réflexe exécuté dans l'intervalle, soit par un des autres processus écrivant dans la mémoire partagée locale de l'application.

Cependant, d'autres styles de calcul sont plus facilement décrits lorsqu'on peut disposer de valeurs à jour des états : par exemple, lorsqu'on veut ajouter ou retrancher une valeur d'un état donné. On peut aussi vouloir détecter facilement le fait qu'un calcul devienne obsolète entre le déclenchement et l'exécution d'un réflexe, comme un calcul de recherche de chemin.

Pour pouvoir utiliser les deux formes de calcul, qui correspondent aux deux manières traditionnelles de traiter les mises à jour de l'état d'une application (*passage de message* et *mise à jour par delta*), les réflexes sont exécutés avec des **copies** des états, effectuées lors du déclenchement de l'ordonnançable, avant leur rattachement à l'ordonnanceur. L'utilisateur précise la manière dont ces copies sont synchronisées avec les états à jour de la mémoire partagée.

Pour définir un ordonnançable, l'utilisateur décrit les états dont il a besoin pour le calcul des conditions de déclenchement et le réflexe associé.

Il utilise pour ce faire les deux instructions *flash* et *update*, dont nous allons définir le comportement pour les deux catégories auxquelles peuvent appartenir ces états.

- l'ensemble des états en entrée : ce sont les états utilisés dans le calcul et les attributs des réflexes. Les valeurs de ces états sont copiées localement quand le réflexe est déclenché.
- l'ensemble des états en sortie : ce sont les états modifiés par les réflexes.

Définition 5.2.2 *L'instruction* **flash**, *utilisable dans le code d'un réflexe, réactualise* ***toutes*** *les copies d'états en entrée et en sortie avec les valeurs à jour des états de la mémoire partagée. Cette instruction est récursive sur les sous-états.*

Quand le calcul nécessite des valeurs d'état à jour, le développeur peut utiliser l'instruction *flash*, pour synchroniser les copies des états avec leurs valeurs à jour. Le caractère récursif de cette instruction a été décidé pour simplifier la description des ordonnançables, ainsi que nous le verrons plus loin.

Définition 5.2.3 *L'instruction* **update**, *utilisable dans le code d'un réflexe, affecte la valeur d'une copie d'état en sortie à l'état correspondant de la mémoire partagée. Cette instruction n'est pas récursive sur les sous-états.*

Le développeur est donc responsable de la synchronisation du résultat d'un calcul avec la mémoire partagée.

Selon cette sémantique, l'utilisateur a la garantie qu'entre une instruction *flash* et une instruction *cooperate*, la valeur d'un état ne peut pas être modifiée par l'exécution d'un autre réflexe. Donc, si l'application est conçue afin qu'un état donné ne soit pas directement modifiable par le réseau (comme c'est par exemple le cas pour un état purement local qui n'est pas sujet à une réplication par hôte distant) ou un autre processus extérieur au *framework* et opérant sur la mémoire partagée, la valeur d'un état et la valeur de sa copie locale pour le réflexe correspondent d'un *flash* à un *cooperate*.

Ainsi, notre modèle permet d'utiliser les deux styles de gestion d'état, par *message* ou mise à jour par *delta*, sans trop sacrifier à la simplicité d'utilisation du modèle d'ordonnancement : la sémantique de l'exécution des réflexes est reproductible modulo les événements provenant des tâches gérées par la couche supérieure préemptive.

Nous définissons ainsi un modèle dont le seul mode de communication entre les réflexes se fait par le biais d'une mémoire partagée, ce qui constitue la principale différence avec l'approche réactive des *Fair-Threads*.

5.3 Dynamique de l'application

Dans cette section, nous allons décrire plus précisément comment se construisent les états (définition 4.5.1, page 113) et les ordonnançables (définition 5.2.1, page 134), et comment l'utilisateur doit les modéliser dans le cadre du *framework*.

Pour faciliter la conception de l'application, nous avons défini un modèle de composition des états, qui peut servir intuitivement à décrire la structure des objets du jeu. Certains états sont tout simplement un ensemble de données, mais la plupart d'entre eux refléteront, les objets virtuels qui composent l'application.

5.3.1 Composition d'états

D'un point de vue génie logiciel, les objets du jeu seront représentés dans notre *framework* par des états. L'approche traditionnelle pour la conception des objets du jeu est de les composer à partir d'autres objets du jeu. Afin de faciliter la conception d'une application en définissant des dépendances logiques entre les états qui reflètent la nature des objets du jeu, le *framework* offre la possibilité d'assembler des états en un état composé. Ainsi, les états sont arrangés hiérarchiquement, en un arbre de compositions (on rappelle que selon la définition 4.5.1 page 113 la composition est stricte, un état ne peut pas appartenir à deux états distincts), la racine de cet arbre étant l'état du jeu lui même. Cette hiérarchie est définie par l'utilisateur lors de la conception du jeu.

Il y a deux manières différentes de définir les sous-états d'un état.
- un état, éventuellement lui-même composé ;
- un ensemble dynamique d'états duplicata (que nous étudierons plus en détail au paragraphe 5.3.1.2 page 138).

Nous utiliserons dans la suite le terme de **composant d'état** pour désigner soit un état, soit un ensemble d'états duplicata.

5.3.1.1 Couplage entre état et composant d'état

Du point de vue des réplications, le couplage entre un état et un de ses composants d'état est faible : quand un état est répliqué sur un hôte distant, seules ses données propres sont répliquées : les données propres des sous-états ne sont pas envoyées sur un hôte distant si leurs propres modèles de réplication ne sont pas déclenchés. De manière similaire, l'instruction *update* (voir définition 5.2.3 page 136) ne met à jour que les données propres d'un état, et pas celles de ses composants d'état.

Cependant, du point de vue de la synchronisation de la mémoire des réflexes, le couplage est fort : quand un réflexe défini un état comme étant nécessaire au calcul tous ses sous-états sont également considérés comme étant nécessaires. Ainsi, lorsque l'instruction *flash* (voir définition 5.2.2 page 135) est invoquée pour un état, la mise à jour est effectuée sur tout l'arbre des sous-états de la copie locale. Cela permet de faciliter la description d'un réflexe opérant sur une hiérarchie d'états représentant un objet du jeu.

5.3.1.2 Duplicata et patrons d'états

A côté des objets persistants d'un monde virtuel, il existe un grand nombre d'objets créés et détruits souvent, en nombre imprévisible, et répondant à la même description et aux même propriétés de réplication.

Par exemple :
- les états représentant les différentes machines clientes sur le serveur
- un ensemble de personnages non-joueurs
- les balles tirées par une arme

Pour représenter plus simplement de tels objets, nous introduisons les notions de *patrons d'états* et d'*états duplicata*.

L'utilisateur du *framework* peut définir une famille d'états en donnant un *patron d'états*. Les états crées à l'aide de ces patrons sont appelés *états duplicata*. Un *patron d'états* défini la structure et les ordonnançables des *états duplicata* qu'il permet de créer.

Ainsi, plusieurs états peuvent être créés et avoir des cycles de vie indépendants en suivant la même description fournie par l'utilisateur. Ils ont la même structure et les mêmes ordonnançables. Ils ont également le même état

parent dans la hiérarchie des états : ce ne sont pas simplement des instances différentes d'un même état. Les états duplicata ne partagent aucune donnée et aucun sous-état : nous avons en effet décrit plus tôt comment les états sont arrangés hiérarchiquement en un arbre.

Si les états duplicata sont bien des états au sens de la définition 4.5.1 (page 113), les ensembles d'états duplicata utilisés pour définir leur appartenance à un état composé n'en sont pas. Ils ont cependant des propriétés en commun, dues au fait que ce sont des composants d'états et peuvent donc être utilisés dans la définition des ordonnançables. Nous commettrons donc parfois l'abus de langage.

Lorsqu'on a défini un ordonnançable pour un *patron d'états* cet ordonnançable sera utilisé par tous les états duplicata issus de ce patron.

5.3.2 Dynamique des états

Les états peuvent être créés ou détruits dynamiquement à travers l'utilisation d'une *fabrique d'états* intégrée au *framework*, et qui peut être utilisée par le développeur dans un réflexe, ou déclenchée par une réplication d'un hôte distant lorsque l'hôte local l'y autorise. La fabrique d'états utilise les descriptions des états fournies par le développeur, qui incluent la structure de l'état, ses ordonnançables et la manière dont il s'initialise et se détruit.

5.3.2.1 Création d'états

La fabrique d'états est capable de créer un état selon une description locale ou distante. La création des éventuels sous-états durant la création de l'état lui-même n'est pas automatique, et est définie par l'utilisateur.

Définition 5.3.1 *Quand un état est créé, tous ses ordonnançables sont dits* **instanciés**.

5.3.2.2 Destruction d'états

Quand un état est détruit, tous ses ordonnançables sont détruits.

Par défaut, les sous-états sont également détruits s'il y en a. Cependant, l'utilisateur peut définir un autre comportement lors de la destruction d'un état dans la fabrique d'états. Il devra alors préciser à quel nouvel état les sous-états non détruits devront être attachés en remplacement de l'ancien parent dans l'arbre hiérarchique des états (nous n'avons cependant pas encore intégré cette possibilité dans le prototype du modèle que nous présenterons ultérieurement).

5.3.3 Dynamique des ordonnançables

On définit deux classes d'états que l'utilisateur doit décrire quand il veut construire un ordonnançable :

Définition 5.3.2 *Les* **états utiles** *d'un ordonnançable sont les* composants d'états *(états ou ensembles d'états duplicata) intervenant dans la description de la temporalité (condition de déclenchement), du code du réflexe de cet ordonnançable, et des éventuels autres attributs de l'ordonnançable (portée et propriétés de communication, dans le cas où l'ordonnançable est un modèle de réplication). L'instruction* flash *(définition 5.2.2, page 135) met à jour les copies de tous les états utiles d'un ordonnançable ainsi que leurs sous-états.*

L'utilisateur peut ainsi simplifier la définition des états utiles d'un ordonnançable en jouant sur la composition des états.

Définition 5.3.3 *Les* **états observés** *d'un ordonnançable sont les composants d'états (états ou ensembles d'états duplicata) dont la mise à jour (par l'instruction* update*, définition 5.2.3, page 136, par une réplication provenant d'un hôte distant autorisé, ou par un ajout ou destruction d'état duplicata dans le cas des ensembles d'états duplicata) provoque un test de vérification de la temporalité (condition de déclenchement) d'un ordonnançable.*

Nous pouvons désormais définir les notions d'**activation** et de **déclenchement** d'un ordonnançable :

Définition 5.3.4 *Un ordonnançable* instancié *est dit* **activé** *si et seulement si au moins un de ses états observés est créé et si tous ses états utiles sont créés.*

Lorsqu'un état est créé, ses ordonnançables ne seront activés qu'une fois ces conditions remplies. Alors seulement ils pourront être déclenchés, provoquant l'exécution de leurs réflexes par l'ordonnanceur, lorsque leurs temporalités seront vérifiées.

Définition 5.3.5 *Un ordonnançable* activé *est* **déclenché** *quand un de ses états observés a été mis à jour, et que sa temporalité (condition de déclenchement) est vérifiée. L'ordonnançable appelle alors implicitement l'instruction* flash *(définition 5.2.2, page 135) pour produire une copie locale de ses états utiles. Si l'ordonnançable est un modèle de réplication (définition 4.5.6, page 116), il procède à la réplication de l'état auquel il est attaché. L'ordonnançable ajoute ensuite son réflexe à la file d'attente de l'ordonnanceur.*

Lorsqu'on a défini un ordonnançable pour un patron d'états et qu'il est **activé**, il sera **déclenché** pour chaque état duplicata quand sa temporalité sera vérifiée.

5.4 Sémantique opérationnelle

Dans cette section, nous allons donner une formalisation partielle des règles sémantiques auxquelles se conforme le modèle que nous avons défini. Nous nous concentrerons sur la description de l'évolution des contextes d'exécution sur chaque hôte de l'application.

5.4.1 Sémantique de calcul dans l'ordonnanceur équitable

Nous allons ici décrire plus formellement comment les instructions utilisateur fournies par le *framework* et utilisables dans le code des réflexes des ordonnançables influent sur le déroulement de l'application sur un hôte de la distribution. Nous utilisons des notations inspirées du calcul des séquents et du calcul propositionnel du premier ordre pour décrire les règles d'inférences qui permettent de modifier l'application à l'aide des instructions dont dispose l'utilisateur pour écrire le code spécifique des réflexes des ordonnançables.

5.4.1.1 Notations

Mémoire partagée M : ensemble des états (voir définition 4.5.1, page 113 créés, valorés, représentant l'état de l'application sur l'hôte courant. Nous noterons $M : s$ pour mettre en valeur l'appartenance de l'état s à M.

Contextes d'exécution des réflexes C_i : mémoires locales des réflexes (voir définition 4.5.5 page 115) en cours d'exécution. Ils correspondent aux copies des états utiles (voir définition 5.3.2, page 140) de l'ordonnançable (voir définition 5.2.1, page 134) auquel le réflexe est attaché. Les C_i sont des ensembles d'états valorés, éventuellement par la valeur *null*. Nous noterons $C_i : s$ pour mettre en valeur l'appartenance de l'état s à C_i.

Suite des contextes des réflexes en cours d'exécution S : suite des contextes d'exécution des réflexes actuellement en attente dans la file de l'ordonnanceur équitable. Quand nous voudrons aller plus dans le détail, nous noterons $\sum_{i=0}^{n} C_i$ pour S.

Ordonnançable $O(s\,, (s_{o,i})_{i=0}^{n}\,, (s_{u,i})_{i=0}^{m}, c)$: un ordonnançable attaché à l'état s, dont les états observés sont les $s_{o,i}$ (voir définition 5.3.3, page 140), dont les états utiles (voir définition 5.3.2, page 140) sont les $s_{u,i}$, et dont la condition de réalisation sur les états (temporalité, voir définiton 4.5.3, page 114) est c. Nous abrégerons parfois cette notation en $O((s_{o,i})_{i=0}^{n}\,, (s_{u,i})_{i=0}^{m})$, voire tout simplement O lorsque le détail ne sera pas nécessaire.

Ensemble des ordonnançable instanciés OI : l'ensemble des ordonnançables instanciés, mais pas encore activés (voir définition 5.3.1, page 139). Nous noterons $OI : O$ pour mettre en valeur l'appartenance de l'ordonnançable O à OI.

Ensemble des ordonnançables activés OA : l'ensemble des ordonnançables activés (voir définition 5.3.4, page 140).Nous noterons $OA : O$ pour mettre en valeur l'appartenance de l'ordonnançable O à OA.

5.4.1.2 Conventions

- Si $s \epsilon M$, et que le contexte d'exécution C_i possède une copie de l'état s, on notera cette copie s^i. On dira alors que les états s et s^i sont **corrélés** ;
- Si $s \epsilon M$ et $s^i \epsilon C_i$ sont corrélés, on notera leurs sous états respectifs s_j et s_j^i de telle sorte que les couples (s_j, s_j^i) soient corrélés.
- Si O_i est un ordonnançable, on notera C_i le contexte d'exécution du réflexe associé à son déclenchement.

5.4.1.3 Définitions

Définition 5.4.1 Affectation simple : *On définit l'opération d'affectation* $[s \leftarrow t]$ *sur deux états corrélés, qui copie les valeurs des données propres de l'état t dans l'état s. Cette opération n'est pas récursive sur les sous-états.*

Cette définition nous servira notamment pour décrire l'effet de l'instruction *update* (voir définition 5.2.3, page 136) dont nous rappelons qu'elle n'est pas récursive sur les sous-états, le couplage entre états et sous états étant faible du point de vue des mises à jour.

Définition 5.4.2 Affectation composée : *On définit l'opération récursive d'affectation composée* σ_s *pour tout état* $s \epsilon M$*, sur le domaine des états corrélés à s telle que :*

- $\sigma_s(s^i) = [s^i \leftarrow s]$ *si s n'a pas de sous-états et* $s^i \neq null$.
- $\sigma_s(s^i)$ *crée une copie de s dans* s^i *si* $s^i = null$
- $\sigma_s(s^i) = [s^i \leftarrow null]$ *si s n'existe pas* ($s \notin M$)*. Ce cas peut se produire par exemple si s a été détruit depuis la création du contexte lors du déclenchement de l'ordonnançable.*
- $\sigma_s(s^i) = [s^i \leftarrow s] \circ (\sigma_{s_j}(s_j^i))_{j=0}^n$ *où les* s_j *sont les sous-états (existants ou non dans M) de s, et les* s_j^i *les sous-états corrélés correspondant des* s^i.

Cette définition nous servira notamment pour décrire l'effet de l'instruction *flash* (voir les définitions 5.2.2, page 135 et 5.3.2, page 140) qui met à jour récursivement toutes les copies d'états du contexte d'un ordonnançable. On rappelle que les états sont strictement organisés en arbres.

Définition 5.4.3 Prédicat de test de déclenchement d'un ordonnançable : *on définit le prédicat* $V(s, M, O)$ *sur la mémoire locale de l'application et un ordonnançable* activé *(*$O \in OA$*).* $V(s, M, O(s\,, (s_{o,i})_{i=0}^{n}\,, (s_{u,i})_{i=0}^{m}, c))$ *est vrai si et seulement si* $s \in M$*,* $s \in (s_{o,i})_{i=0}^{n}$ *et si* c *est vérifiée pour les valeurs actuelles de* M*.*

En d'autres termes, un ordonnançable activé est déclenché quand un de ses états observés a été modifié et que sa temporalité est vérifiée.

Définition 5.4.4 Instanciation et activation des ordonnançables lors de la création d'un état s : *on définit les transformations* $\tau_{OI,s}^{M}$*, et* $\tau_{OA,s}^{M}$ *sur les ensembles* OI *et* OA *d' ordonnançables telles que :*

- $\tau_{OI,s}^{M}(OI)$ *ajoute à* OI *tous les ordonnançables* $O((s_{o,i})_{i=0}^{n}\,, (s_{u,i})_{i=0}^{m})$ *attachés à l'état* s *tels que :*
  $(\forall i\ 0 \leq i \leq n\ s_{o,i} \notin M) \vee (\forall i\ 0 \leq i \leq m\ \exists s_{u,i} \notin M)$
- $\tau_{OA,s}^{M}(OA)$ *ajoute à* OA *tous les ordonnançables* $O((s_{o,i})_{i=0}^{n}\,, (s_{u,i})_{i=0}^{m})$ *attachés à l'état* s *tels que :*
  $(\forall i\ 0 \leq i \leq n\ \exists s_{o,i} \in M) \wedge (\forall i\ 0 \leq i \leq m\ s_{u,i} \in M)$

Autrement dit, en utilisant la définition d'un ordonnançable activé (définition 5.3.4, page 140) : la transformation $\tau_{OI,s}^{M}$ ajoute à OI tous les ordonnançables de l'état s qui ne sont pas activés : aucun de leurs états observés n'existe ou il leur manque au moins un de leurs états utiles. La transformation $\tau_{OA,s}^{M}$ ajoute à OA tous les ordonnançables de s activés : au moins un de leurs états observés existe, et tous leurs états utiles existent.

Nous utiliserons ces transformations pour exprimer que lors de la création d'un état, tous ses ordonnançables sont instanciés, et activés lorsque les états utiles et observés de l'ordonnançable vérifient les conditions d'activation.

Définition 5.4.5 Destruction des modèles de réplication lors de la destruction d'un état s : *on définit les transformations* $\overline{\tau_s^M}$*, sur les ensembles* OI *et* OA *d'ordonnançables telles que :*

- *Pour tout* $s \in M$*,* $\overline{\tau_s^M}(OI)$ *soustrait de* OI *tous les ordonnançables attachés à* s*.*
- *Même définition pour* OA

Nous utiliserons ces transformations pour exprimer que lors de la destruction d'un état, tous ses ordonnançables sont détruits.

5.4.1.4 Séquent

Le calcul que nous présentons dans cette section permettra de produire le séquent suivant, représentant l'application sur un hôte :

$$\langle M\,;\,OI\,;\,OA\,;\,S\rangle \vdash C_k$$

Le membre gauche représente l'état de l'application (la mémoire partagée M, l'ensemble des ordonnançables instanciés OI, l'ensemble des ordonnançables activés OA, l'ensemble des contextes des réflexes en attente d'exécution ou de reprise d'exécution par l'ordonnanceur S) et le membre droit le contexte du réflexe actuellement en cours d'exécution C_k.

5.4.1.5 Règles d'inférences

Les règles que nous présentons dans cette partie montrent l'évolution du séquent représentant l'état de l'application sur un hôte lors de l'exécution des instructions définies par notre *framework* dans le code des réflexes.

Mise à jour d'un état s par un réflexe : la valeur de l'état s dans le contexte local est affecté à l'état de la mémoire globale s'il y est instancié. La fonction *update* renvoie alors *true*. Dans le cas où l'état n'est plus instancié dans la mémoire M, l'opération n'a aucun effet mais la fonction renvoie *false*.

$$\frac{\begin{array}{c}\langle M : s\,;\,OI\,;\,OA\,;\,S\rangle \vdash C_k : s^k \\ \forall j\; 0 \leq j \leq m\; O_j \,\epsilon\, OA \;\wedge\; V(s, M[s \leftarrow s^k], O_j)\end{array}}{\langle M : [s \leftarrow s^k]\,;\,OI\,;\,OA\,;\,S + \sum_{i=0}^{m} C_i\rangle \vdash C_k : s^k}\ (\text{UPDATE}(s))$$

Cette règle utilise le prédicat de vérification des conditions de déclenchement d'un ordonnançable (définition 5.4.3) et l'affectation simple (définition 5.4.1).

Elle montre comment la mise à jour d'un état peut provoquer le déclenchement d'ordonnançables activés, ajoutant les contextes d'exécution des réflexes qui y sont associés à la file d'attente de l'ordonnanceur.

Coopération : le réflexe actuellement en cours d'exécution rend la main à l'ordonnanceur, qui exécute ou reprend l'exécution du réflexe suivant dans sa file d'attente.

$$\frac{\langle\, M\,;\, OI\,;\, OA\,;\, \sum_{i=0,i\neq k}^{n} C_i \,\rangle \;\vdash\; C_k \qquad j = k+1 \; si \; k < n,\; 0 \; sinon}{\langle\, M\,;\, OI\,;\, OA\,;\, \sum_{i=0,i\neq j}^{n} C_i \,\rangle \;\vdash\; C_j} \quad (\text{COOPERATE})$$

Mise à jour d'un contexte d'exécution de réflexe : l'instruction *flash* met à jour chaque état du contexte d'exécution du réflexe actif. La règle est donnée pour un état, mais elle s'applique à tous les états du contexte de ce réflexe.

$$\frac{\langle\, M\,;\, OI\,;\, OA\,;\, S\,\rangle \;\vdash\; C_k : s^k}{\langle\, M\,;\, OI\,;\, OA\,;\, S\,\rangle \;\vdash\; C_k : \sigma_s(s^k)} \quad (\text{FLASH}(s))$$

Cette règle utilise l'affectation composée (définition 5.4.2) car l'instruction flash est récursive sur les sous-états.

Création d'un état : la création d'un état provoque plusieurs modifications sur le contexte d'exécution global de l'application. L'état est ajouté à la mémoire, et les ordonnançables associés sont ajoutés suivant les transformations $\tau^M_{OI,s}$ et $\tau^M_{OA,s}$ (définition 5.4.4) aux ensembles d'ordonnançables OI ou OA.

Les règles suivantes illustrent l'impact de la création d'un état sur les ordonnançables n'étant pas attachés à l'état créé : un ordonnançable instancié est activé si au moins un de ses états observés est instancié, et si tous les états nécessités sont instanciés. Il est désactivé lorsque l'une de ces conditions n'est plus requise.

$$\frac{\begin{array}{c}\langle\, M\,;\, OI : O((s_{o,i})_{i=0}^{n}\,,\,(s_{u,i})_{i=0}^{m})\,;\, OA\,;\, S\,\rangle \;\vdash\; C_j \qquad s_{o,k} \notin M \\ \forall i\; 0 \leq i \leq m \; s_{u,i} \in M \qquad \forall i\; 0 \leq i \leq n \; s_{o,i} \notin M\end{array}}{\langle\, M : s_{o,k}\,;\, \tau^M_{OI,s_{o,k}}(OI)\,;\, \tau^M_{OA,s_{o,k}}(OA) : O((s_{o,i})_{i=0}^{n}\,,\,(s_{u,i})_{i=0}^{m})\,;\, S\,\rangle \;\vdash\; C_j} \quad (\text{CREATE}(s_{o,k}))$$

Cette première règle décrit comment un ordonnançable dont tous les états utiles existent, mais dont aucun état observé n'existe passe de l'état instancié à l'état activé, lors de la création d'un de ses états observés.

$$\frac{\langle\, M\,;\, OI : O((s_{o,i})_{i=0}^{n}\,,\,(s_{u,i})_{i=0}^{m})\,;\, OA\,;\, S\,\rangle \;\vdash\; C_j \qquad \forall i\; 0 \leq i \leq m,\; i \neq k\; s_{u,i} \in M \qquad \exists i\; 0 \leq i \leq n\; s_{o,i} \in M}{\langle\, M : s_{u,k}\,;\, \tau^{M}_{OI,s_{u,k}}(OI)\,;\, \tau^{M}_{OA,s_{u,k}}(OA) : O((s_{o,i})_{i=0}^{n}\,,\,(s_{u,i})_{i=0}^{m})\,;\, S\,\rangle \;\vdash\; C_j}\;(\text{CREATE}(s_{u,k}))$$

Cette seconde règle décrit comment un ordonnançable dont au moins un état observé existe mais dont il manque un seul état utile passe à l'état activé lorsque l'état utile manquant est créé.

Remarque : selon la définition de la fabrique de l'état, des sous-états peuvent être également créés à la suite de ces opérations. Les mêmes règles s'appliquent alors.

Destruction d'un état : La destruction d'un état retire cet état de la mémoire, et provoque le retrait des ordonnançables associés suivant la transformation $\overline{\tau_s^M}$ (définition 5.4.5, page 144).

Les règles suivantes sont les duales de celles régissant la création d'état, et illustrent l'impact de la destruction d'un état sur les ordonnançables n'étant pas attachés à l'état créé : un ordonnançable n'est plus activé si il n'existe plus aucun de ses états observés, ou si il manque un de ses états utiles.

$$\frac{\langle\, M : s_{o,k}\,;\, OI\,;\, OA : O((s_{o,i})_{i=0}^{n}\,,\,(s_{u,i})_{i=0}^{m})\,;\, S\,\rangle \;\vdash\; C_j \qquad \forall i\; 0 \leq i \leq m,\; i \neq k\; s_{o,i} \notin M}{\langle\, M\,;\, \overline{\tau^{M}_{s_{o,k}}}(OI) : O((s_{o,i})_{i=0}^{n}\,,\,(s_{u,i})_{i=0}^{m})\,;\, \overline{\tau^{M}_{s_{o,k}}}(OA)\,;\, S\,\rangle \;\vdash\; C_j}\;(\text{DESTROY}(s_{o,k}))$$

Cette première règle décrit qu'un ordonnançable activé dont l'état détruit était le seul état observé est désactivé.

$$\frac{\langle\, M : s_n^k\,;\, OI\,;\, OA : O((s_{o,i})_{i=0}^{n}\,,\,(s_{u,i})_{i=0}^{m})\,;\, S\,\rangle \;\vdash\; C_j : s_{u,k}^{j}}{\langle\, M\,;\, \overline{\tau^{M}_{s_{u,k}}}(OI) : O((s_{o,i})_{i=0}^{n}\,,\,(s_{u,i})_{i=0}^{m})\,;\, \overline{\tau^{M}_{s_n^k}}(OA)\,;\, S\,\rangle \;\vdash\; C_j : [s_{u,k}^{j} \leftarrow null]}\;(\text{DESTROY}(s_{u,k}))$$

Cette seconde règle décrit qu'un ordonnançable activé dont un état utile est détruit est désactivé.

Remarque : les mêmes règles s'appliquent à la destruction des sous-états.

5.4.2 Sémantique de l'application distribuée

Nous allons ici décrire comment les processus qui représentent chaque hôte de l'application distribuée interagissent.

5.4.2.1 Notations

Ensembles notables :

- M est la mémoire locale de chaque processus, qui représente l'état de l'application sur un hôte de la distribution. Nous noterons s un état (définition 4.5.1, page 113) de M
- $R_a(M)$ est l'ensemble des modèles de réplication activés (définition 5.3.4, page 140) de M.
- $A(M, P)$ est l'ensemble des états de M dont la mise à jour est autorisée par le processus distant P.
- F est l'ensemble des *fabriques* (voir section 5.3.2 page 139) d'états existantes dans chaque processus. Nous noterons $f_s \in F$ la fabrique f_s de l'état s.
- $A(F, P)$ est l'ensemble des *fabriques* de F dont l'utilisation pour la création d'un état est autorisée de manière distante pour le processus P.
- S est un ensemble fini de processus distants, que nous noterons parfois $\bigcup_{i=0}^{n} P_i$ lorsque nous voudrons entrer dans le détail.
- $\prod$ l'ensemble des processus de l'application distribuée.

Événements : Les processus représentant les différents hôtes de l'application réagissent aux événements suivants :

- $s(P)$: réception de la valeur d'un état $s \in M$, M étant la mémoire locale du processus courant, provenant du processus P.
- $f_s(P)$: réception de la valeur d'un état $s \notin M$, M étant la mémoire locale du processus courant, tel que $f_s \in F$, provenant du processus P.

– $R(s, S)$: la réplication de la valeur de l'état $s \in M$, M étant la mémoire locale du processus courant, à l'ensemble de processus destinataires S. Cet événement est provoqué en interne par le processus courant. Il est produit lors de l'application de la règle (UPDATE), décrite dans la section 5.4.1 présentant la sémantique de calcul local de l'ordonnanceur, lorsqu'une mise à jour provoque la réalisation des conditions d'activation d'un modèle de réplication. Nous abrégerons souvent cette notation en $R(s)$ lorsque le détail ne sera pas nécessaire.

5.4.2.2 Représentation des processus

Nous utiliserons un formalisme inspiré par le calcul CCS [69] :
– la notation $P \quad -t$ signifie que le processus P reçoit l'événement t.
– la notation $P \quad | \quad P'$ signifie que les processus P et P' s'exécutent en parallèle. Ils représentent deux hôtes différents de l'application.
Nous décrivons chaque processus participant à l'application sous la forme :

$$P' = \sum_{s \in M,\, P \in \prod} s(P).Q_s + \sum_{R(s) \in R_a(M), s \in M} R(s).P' + \sum_{f_s \in F,\, P \in \prod} f_s(P).Q_{f_s}$$

où :
– pour chaque état $s \in M$, Q_s est le processus P' où la valeur de s a été mise à jour dans la mémoire M à l'aide de la valeur reçue par l'événement $s(P)$, d'une manière identique à la règle (UPDATE(s)) décrite dans la section 5.4.1 présentant la sémantique de calcul local de l'ordonnanceur.
– pour chaque fabrique $f_s \in F$, Q_{f_s} est le processus P' où l'état s a été ajouté à la mémoire M, mis à jour avec la valeur reçue par l'événement $f_s(P)$, et dont les ensembles d'ordonnançables ont été mis à jour, d'une manière identique aux règles (CREATE(s)) décrites dans la section 5.4.1 présentant la sémantique de calcul local de l'ordonnanceur.

Les règles d'inférences définies dans la suite de cette section permettent d'exprimer la signification informelle suivante de cette description :

Chaque processus P', représentant un hôte de l'application distribuée à un moment donné de son exécution, peut recevoir un événement $s(P)$ et devenir le processus Q_s, ou un événement $R(s)$ et rester le processus P', ou un événement f_s et devenir le processus Q_{f_s}.

Nous abrégerons parfois cette notation pour alléger l'écriture des règles par :

$$P' = \sum s(P).Q_s + \sum R(s).P' + \sum f_s(P).Q_{f_s}$$

ou même :

$$P' = P'(M, F)$$

5.4.2.3 Règles d'inférence

Mise à jour d'un état par un processus distant : cette règle décrit la réaction d'un processus P' à la réception de la mise à jour d'un état instancié envoyé par un processus distant :

$$\frac{\sum_{s \in M,\, P \in \prod} s(P).Q_s \;+\; \sum R(s).P' \;+\; \sum f_s(P).Q_{f_s} \qquad - s(P)}{Q_s} \quad s \in A(M,P)$$

(UPDATE REP(s))

Création d'un état par un processus distant : cette règle décrit la réaction d'un processus P' à la réception d'un état non instancié envoyé par un processus distant :

$$\frac{\sum s(P).Q_s \;+\; \sum R(s).P' \;+\; \sum_{f_s \in F,\, P \in \prod} f_s(P).Q_{f_s} \qquad - f_s(P)}{Q_{f_s}} \quad f_s \in A(F,P)$$

(CREATE REP(s))

Réplication de la valeur d'un état : ces deux règles décrivent comment la réplication sur un hôte de la distribution propage l'information aux autres processus distants selon la portée de la réplication. Il est à noter que comme il s'agit de transmission par le réseau, l'application de ces règles n'est valide que dans le cas où le processus distant reçoit effectivement les informations : dans le cas où les propriétés de communication de la réplication ne sont pas

fiable, le message ne sera pas reçu et les règles ne s'appliquent donc pas.

$$\frac{\sum s(P).Q_s + \sum_{R(s)\in R_a(M), s\in M} R(s).P' + \sum f_s(P).Q_{f_s} - R(t,S) \mid P''(M'',F'') \qquad t \in M'',\ P'' \in S}{P' \mid P'' - t(P')} \text{(REPLICATE UPDATE}(s)\text{)}$$

Lorsqu'un processus émet une réplication, il n'est pas modifié mais tous les processus destinataires et pour lesquels l'état est créé reçoivent une mise à jour de l'état, si elle est autorisée pour l'émetteur.

$$\frac{\sum s(P).Q_s + \sum_{R(s)\in R_a(M), s\in M} R(s).P' + \sum f_s(P).Q_{f_s} - R(t,S) \mid P''(M'',F'') \qquad t \notin M'',\ f_t \in F'',\ P'' \in S}{P' \mid P'' - f_t(P')} \text{(REPLICATE CREATE}(s)\text{)}$$

Lorsqu'un processus émet une réplication, il n'est pas modifié mais tous les processus destinataires et pour lesquels l'état n'est pas créé reçoivent une création d'état, si elle est autorisée pour l'émetteur.

5.5 Recettes de cuisine

Nous donnons ici quelques utilisations typiques de notre modèle pour des situations fréquemment rencontrées.

Comment obtenir une exécution reproductible Les facteurs susceptibles d'introduire de la non-reproductibilité dans l'application sont les tâches de la couche supérieure préemptive de notre modèle.

Dans le cas où aucune bibliothèque externe au *framework* n'est utilisée, ces tâches sont donc principalement les modifications d'états provenant de réplications par des hôtes distant, ou les modifications d'états provenant des commandes de l'utilisateur de l'application.

Pour mettre au point l'application, on peut donc scénariser ces mises à jour. De même, si une erreur se produit, pour peu qu'une trace soit gardée des

événements réseau et commandes utilisateur, il est possible de la reproduire et donc de faciliter sa correction.

Comment terminer précocement un calcul devenu obsolète Un réflexe peut correspondre à un calcul lourd dont le résultat peut devenir obsolète avant qu'il ne soit terminé : par exemple, un calcul de *dead-reckoning* entamé devient finalement inutile quand une nouvelle mise à jour de la position extrapolée arrive.

Il peut être judicieux de dédier un état particulier pour vérifier si le calcul en cours est toujours valide. Dans notre exemple, la mise à jour de la position peut être suivie par le déclenchement d'un ordonnançable modifiant cet état. Le réflexe correspondant au calcul de *Dead-Reckoning* peut utiliser régulièrement l'instruction *flash* afin de terminer le calcul s'il est devenu obsolète.

Comment donner une priorité d'accès aux ressources à un traitement particulier Le modèle que nous avons décrit étant équitable, cette notion n'a de sens qu'en ce qui concerne le temps de calcul alloué à chaque réflexe. On ne peut donc jouer qu'avec la réduction du nombre des instructions *cooperate*.

Comment modéliser une exécution de réflexe du style message Il suffit de ne jamais utiliser l'instruction *flash*, afin de ne pas perdre la valeur initiale de l'état représentant le message.

Comment s'assurer qu'un état concerné par un réflexe s'exécute avec une valeur à jour pour cet état La solution est de l'ordre de la conception des états de l'application. Nous avons vu que selon la sémantique de l'ordonnancement des réflexes, on a la garantie qu'un état ne peut pas être modifié par un autre réflexe entre une instruction *flash* et une instruction *cooperate*.

Il suffit de définir l'état en question de façon à ce qu'il soit local à l'application, ne faisant pas partie de l'état global du jeu et donc non susceptible d'être modifié directement par une réplication distante, et qu'il ne puisse

également pas être modifié par les autres processus de la couche supérieure préemptive.

Cette définition ne réduit pas le champ des possibilités, puisque l'état pourra être indirectement modifié par des réplications distantes, mais en utilisant un ordonnançable réagissant à la modification d'un autre état répliqué : l'ordonnançable effectuant la modification de l'état et déclenché par la modification d'un autre état obtenu par réplication passera alors par l'ordonnanceur équitable des réflexes.

Chapitre 6

Implémentation

Le *Fill-In-The-Gaps Toolkit* est un prototype destiné à illustrer notre approche. Il implémente le modèle de comportement d'une application distribuée selon la sémantique étudiée au chapitre précédent, et peut se découper en plusieurs parties.

- Un *calcul local*, comprenant l'ordonnanceur travaillant sur des copies d'états, exécute les réflexes des ordonnançables selon la sémantique que nous avons définie. Les réflexes sont décrits par l'utilisateur dans le langage Java, qui est aussi le langage de l'implémentation. L'utilisateur dispose également des instructions *flash* (mise à jour du contexte d'exécution local), *update* (mise à jour des données propres d'un état), *cooperate* (interruption d'exécution temporaire), *create* (création d'un état) et *destroy* (destruction d'un état).
- Un ensemble hiérarchisé de classes abstraites et d'interfaces permet de définir les *composants* de l'application : états, ensembles d'états duplicata et ordonnançables. Cet ensemble permet également de définir les briques de base nécessaires à la constitution des modèles de réplication, les portées et les propriétés de communication.
- La *carte de l'application*, rassemble tous les composants de l'application ainsi que l'état local du jeu.

Nous avons défini également quelques éléments de départ pour une bibliothèque de briques de base (les *Gaps* du *Fill-In-The-Gaps Toolkit*) afin de pouvoir montrer comment utiliser le *framework*.

Nous avons raffiné notre prototype jusqu'à en obtenir une version qui prend en entrée des éléments pouvant être générés automatiquement à partir d'une description scriptée de l'assemblage des états et des ordonnançables, et du code des réflexes. Nous montrons ainsi comment le *framework* peut s'intégrer ultérieurement dans un outil futur.

Ce chapitre présente ce travail. Après avoir expliqué le choix du langage de l'implémentation, nous montrons comment le développeur peut actuellement l'utiliser, en expliquant quels sont les aspects que nous pensons automatiser à partir d'un outil de développement futur. Nous dédions ensuite trois sections à la présentation de la façon dont nous avons implémenté les trois ensembles fonctionnels du framework : l'ordonnanceur, la hiérarchie des composants, et la carte de l'application. Pour finir, nous donnons un exemple simple d'application réalisé à l'aide du framework et de quelques éléments de la bibliothèque associée.

6.1 Un prototype en Java

Nous avons choisi de développer le prototype en Java. Les raisons de ce choix sont tout d'abord que l'auteur de cette thèse apprécie ce langage. Mais il y a aussi des raisons moins subjectives.

Tout d'abord, les outils de développement pour ce langage, et ses bibliothèques très intégrées en font en langage de choix pour un prototypage rapide, à but démonstratif. Le *ramasse-miette*, même s'il n'est pas une arme ultime contre les fuites mémoires, permet un grand confort de programmation. Nous avons eu besoin des qualités d'introspection du langage, et avons avantageusement profité de son modèle orienté objet pour factoriser du code, et mettre à profit polymorphisme et liaison tardive.

Ce choix n'a donc pas été dicté, comme souvent, par des impératifs de portabilité : en effet, la bibliothèque des *threads* de Java est très complexe à utiliser si on veut réaliser une application portable. Comme cela n'était pas notre objectif, nous sommes partis du principe que nous disposions d'une machine virtuelle utilisant un modèle de *thread* préemptif. De même, nous ne nous sommes pas arrêtés aux problèmes éventuels de performances que nous pourrions éventuellement rencontrer en utilisant une stratégie d'ordon-

nancement existante afin de développer notre propre ordonnanceur : le *Fill-In-The-Gaps Toolkit* est un prototype.

6.2 Le point de vue de l'utilisateur

Pour réaliser une application à l'aide du *Fill-In-The-Gaps Toolkit*, l'utilisateur doit assembler les données de l'application en une hiérarchie d'états. Il doit également créer des ordonnançables, en composant simplement des éléments de la bibliothèque, en spécialisant un de ces éléments, ou en en définissant de nouveaux qui s'intégreront à l'application. Nous allons montrer dans cette partie comment l'utilisateur du *framework* doit définir les états et les ordonnançables, ainsi que la manière dont il peut étendre la bibliothèque des blocs de base avec de nouvelles portées et de nouvelles définitions de propriétés de communication.

6.2.1 Décrire un état

Les éléments de code que nous allons présenter ici sont destinés à être, à terme, générés par un outil d'intégration construit autour du *Fill-In-The-Gaps Toolkit*.

Pour décrire un état il faut disposer des trois éléments suivants :
- une classe décrivant l'architecture de l'état : ses données propres, ses sous-états, et ses ensembles d'états duplicata ;
- une classe fabrique, permettant au *framework* de créer des instances de la classe précédente ;
- un texte de configuration qui permettra de paramétrer l'utilisation de la fabrique.

Nous allons décrire dans cette partie les responsabilités de ces différents éléments, et les illustrer par des exemples.

6.2.1.1 Classe de description d'un état

Un état est une instance d'une classe héritant de la classe abstraite `State`, ne l'étendant qu'avec des attributs et leurs accesseurs. Les paramètres du

constructeur de la classe sont à passer au constructeur de la super-classe, et seront fournis par la fabrique.

Les attributs représentant les données propres de l'état sont dans l'implémentation actuelle les types Java `int`, `boolean` et `String`.

Les autres attributs d'un état sont ses sous-états, ou des ensembles d'états duplicata.

Exemple de classe étendant la classe *State* :

```
public class MyState extends State{
    private int x ;
    private int y ;
    private MySubState subState ;
    public MyState(String key, DataAccessList dataAccessList,
            String subStatesKeys ,StateAccessList stateSetterList ,
            String datasInitialization , String replicationKeys ,
            String behaviorKeys , String substateInitialization ,
            StateFactory stateFactory){
        super(key, dataAccessList, subStatesKeys , stateSetterList ,
            datasInitialization , replicationKeys , behaviorKeys ,
            substateInitialization, stateFactory) ;
    }
    public synchronized int getX() {
        return x;
    }
    public synchronized void setX(int x) {
        this.x = x;
    }
    public synchronized int getY() {
        return y;
    }
    public synchronized void setY(int y) {
        this.y = y;
    }
    public synchronized MySubState getSubState() {
        return subState;
    }
    public synchronized void setSubState(MySubState subState) {
```

```
        this.subState = subState;
    }
}
```

6.2.1.2 La fabrique d'états

Afin d'instancier un état, le *framework* utilise des fabriques dédiées, implémentant l'interface `StateFactory`. Cette organisation correspond au patron de conception classique *Fabrique Abstraite* décrite dans [46]. Il y a donc une fabrique par classe de description d'états.

La méthode de création de l'état prend en paramètres les éléments du fichier de configuration décrivant comment l'état doit être créé et qui comprend :

- l'identification de l'instance d'état créée dans l'arborescence de l'application ;
- la liste des identifiants de ses données propres et de leurs valeurs d'initialisation ;
- la liste des identifiants de ses modèles de réplication et des ordonnançables comportementaux qui lui sont associés ;
- la liste des identifiants de ses sous-états, et de ceux d'entre eux qui doivent être instanciés dès la création de l'état ;

Ces éléments sont utilisés pour construire les paramètres du constructeur de la classe `State`.

L'objectif de la fabrique est de fournir à la *carte de l'application*, un des composants du *framework* que nous étudierons plus loin, la définition des sous-états et des ordonnançables de l'état créé.

Il est également de faire le lien entre les identifiants des composants d'un état (ses données propres et ses sous-états) et les accesseurs définis pour ces mêmes composants dans la classe le représentant.

Pour ce faire, la méthode de création de la fabrique doit constituer des associations entre les clés des composants et les méthodes qui permettent d'y accéder. Ce sont les paramètres `DataAccessList dataAccessList` et `StateAccessList stateSetterList` du constructeur de la classe `State`.

Par exemple, la méthode de création de la fabrique de l'état `MyState` comportera donc le code suivant, la classe `DataAccess` définissant l'associa-

tion de l'identifiant d'une donnée propre pour nos types de base avec ses accesseurs, et la classe `StateAccessInState` définissant l'association d'un composant d'état avec ses accesseurs.

Exemple de créations d'associations entre les attributs d'un état et ses accesseurs dans la fabrique d'un état :

```
DataAccessList dataAccessList =  new DataAccessList() ;
// association de l'identifiant de l'attribut x de type int
// avec ses accesseurs
dataAccessList.put(idX, new DataAccess(DataAccess.INTEGER_TYPE){
    protected void setData(State state, int value) {
       ((MyState)state).setX(value) ;
    }
    protected int getInt(State state){
        return ((MyState)state).getX() ;
    }
}) ;
// association de l'identifiant de l'attribut y de type int
// avec ses accesseurs
dataAccessList.put(idY, new DataAccess(DataAccess.INTEGER_TYPE){
    protected void setData(State state, int value) {
       ((MyState)state).setY(value) ;
    }
    protected int getInt(State state){
        return ((MyState)state).getY() ;
    }
}) ;

StateAccessList stateSetterList = new StateAccessList() ;
// association de l'identifiant de l'attribut subState de type
// MySubState avec ses accesseurs
stateSetterList.put(idSubstate,new StateAccessInState() {
    public StateComponent getState(StateCreationObserver
                          stateCreationObserver) {
        return ((MyState)stateCreationObserver).getSubState();
    }
    public void setState(StateCreationObserver
                         stateCreationObserver, StateComponent state) {
```

```
        ((MyState)stateCreationObserver).setSubState((MySubState)
                                                        state) ;
    }
}) ;
```

Ces associations seront ensuite utilisées par le *framework* pour répercuter les changements d'état local de l'application lorsqu'elles proviennent d'un hôte distant, ou lors de la création des sous-états.

Le code des fabriques d'états doit également être à terme généré par un environnement de développement futur.

6.2.1.3 Paramétrer l'utilisation de la fabrique d'un état à l'aide d'un texte de configuration :

Les paramètres utilisés par la méthode de création de la fabrique d'un état sont issus d'un fichier de configuration fourni par l'utilisateur. Nous en donnons un exemple qui correspond à ceux que nous avons présentés pour la classe de description de l'état et sa fabrique :

```
STATE stateKey stateFactoryKey
KEYSUBSTATE substateKey KEYDATA xKey yKey DATAINITIALIZE xKey.100.int
yKey.200.int
SUBSTATEINITIALIZE substateKey
REPLICATIONMODELS stateChangedReplication
```

Ce texte de configuration décrit un état :
- dont l'identifiant est `stateKey`, et dont l'identifiant de la fabrique abstraite est `stateFactoryKey` ;
- qui a comme attributs le sous-état d'identifiant `substateKey` et les données propres d'identifiants `xKey` et `yKey` ;
- dont les données propres sont initialisées à la création de l'état par les valeurs respectives 100 et 200 et sont de type `int` ;
- dont le sous-état d'identifiant `substateKey` devra être créé également lors de la création de l'état `stateKey` ;
- qui a un modèle de réplication, dont l'identifiant est `stateChangeReplication`

6.2.2 Décrire les ordonnançables

La description des ordonnançables repose exactement sur le même principe que la description d'un état. Nous n'allons donc pas rentrer dans les détails, un exemple complet de modélisation étant exposé à la fin de ce chapitre.

L'utilisateur doit décrire une classe concrète étendant les classes abstraites `Replication` (pour les réplications) ou `Behavior` (pour les ordonnançables sans réplication associée). Ces classes concrètes ont comme attributs les références sur les états utiles (définition 5.3.2, page 140) de l'ordonnançable, et fournissent leurs accesseurs. Le code de l'éventuel réflexe est défini par l'utilisateur dans le constructeur de l'ordonnançable. La condition de réalisation est une méthode abstraite des classes `Replication` et `Behavior` à redéfinir par la classe de description de l'ordonnançable.

Il faut également disposer d'une fabrique, étendant une fabrique abstraite, qui utilisera un texte de configuration pour construire les associations permettant d'identifier les états utiles pour le calcul des conditions et des réflexes.

Le texte de configuration permet également à la fabrique de passer au constructeur de la classe de description de l'ordonnançable l'état auquel il est attaché, les identifiants de ses états observés (définition 5.3.3 page 140), ainsi que les identifiants des éventuelles portées et propriétés de communication dans le cas de la fabrique d'un modèle de réplication.

L'exemple suivant est une classe de description d'un modèle de réplication, qui a comme état utile une instance de la classe `MyState`. Pour créer un modèle de réplication de cette classe, la fabrique utilisera le fichier de configuration pour savoir à quel état de la mémoire partagée cet attribut est lié, et créer un objet l'associant avec sa méthode d'affectation comme dans le cas des fabriques d'états. Cette association sera utilisée par le *framework* lors de l'instanciation du modèle de réplication, immédiatement si l'état en question est déjà créé, lors de sa création sinon.

```
public class MyStateReplication extends Replication {
    private MyState myState;
    public MyStateReplication(State state , StateAccessList stateSetters ,
        String scope , String protocol , String statesObserved) {
```

```
        super(state , stateSetters , scope , protocol , statesObserved);
        //reflexe de l'ordonnancable
        super.callback = new Callback(this) {
            protected void execute() {
                myState.setParam(myState.getParam() + 2) ;
                updateState(myState);
            }
        };
    }
    public synchronized void setMyState(MyState myState) {
        this.myState = myState;
    }
    public synchronized MyState getMyState() {
        return myState;
    }
    // condition de réalisation, temporalité, de l'ordonnancable
    // ici, qu'un etat observe est modifie
    protected boolean realized() {
        return true ;
    }
}
```

Comme pour les états, nous envisageons à terme de générer tout le code (hors réflexe et condition de réalisation) des ordonnançables et de leurs fabriques.

La bibliothèque de composants peut également proposer des classes décrivant des ordonnançables : par exemple, un modèle de réplication observant seulement l'état qu'il réplique, et qui est vérifié chaque fois que ce dernier est modifié. Elle peut également être étendue par des ordonnançables permettant de décrire des comportements autonomes des objets du monde virtuel.

6.2.3 Décrire une portée

L'utilisateur peut définir de nouvelles portées dans la bibliothèque des briques de base associée au *framework*.

La définition d'une portée repose également sur le principe de l'association d'une classe de description, étendant la classe abstraite `Scope` fournie par le *framework*, d'une fabrique abstraite, et d'un texte de configuration

permettant d'utiliser ces classes lors de l'intégration de l'application.

Mais dans le cas de la description d'une portée, la fabrique se contente d'en passer l'identifiant au constructeur de la classe. Le texte de configuration est donc également très simple, associant un identifiant pour l'instance de la classe à l'identifiant de la fabrique permettant de la construire :
`SCOPEDEF scopeKey scopeFactoryKey`

La classe abstraite `Scope` dispose d'une référence vers l'ensemble des hôtes connus de l'application locale.

La classe de description d'une nouvelle portée doit utiliser cette référence. Elle doit définir la méthode abstraite `send`, qui prend en paramètre le modèle de réplication qui utilise la portée, l'état à répliquer, et les propriétés de communication à utiliser.

Dans le code de cette définition, l'utilisateur doit sélectionner les hôtes à qui la réplication doit être transmise, selon leurs caractéristiques, puis leur déléguer l'envoi de l'état avec les propriétés de communication qu'il a reçues en paramètre.

Voici un exemple simple de classe dont une instance représentera une portée qui est l'ensemble des hôtes distants connus de l'application :

```
public class AllScope extends Scope{
    public AllScope(String key) {
        super(key);
    }
    public void send(Replication replication , State state,
                     CommunicationProperties protocol) {
        for(int i = 0 ; i<hosts.nbRecipients() ; i++){
            Recipient recipient =  hosts.getRecipientByRank(i) ;
            recipient.send(state,protocol) ;
        }
    }
}
```

6.2.4 Décrire une propriété de communication

Si l'utilisateur veut étendre la bibliothèque des blocs de base avec de nouvelles propriétés de communication, il doit comme dans le cas précédent définir une nouvelle classe de description et une fabrique associée permettant de créer une instance de cette classe en passant un identifiant à son constructeur. De même que précédemment, lors de l'intégration de l'application, le texte de configuration permettant d'utiliser cette nouvelle brique de base associera l'identifiant de l'instance à l'identifiant de la fabrique permettant de la créer :

```
PROTOCOLDEF protocolKey protocolFactoryKey
```

Pour décrire une nouvelle propriété de communication, l'utilisateur doit fournir une classe de description qui étend la classe abstraite `CommunicationProperties`.

Il doit alors définir trois méthodes abstraites :

- la méthode `processMessage` prend en paramètre l'état à répliquer, et doit procéder aux traitements avant envoi du message (cryptage, découpage en plusieurs paquets à envoyer séparément...). Cette méthode renvoie une liste de chaînes de caractères.
- la méthode `send` prend en paramètre une liste de chaînes de caractères, et une classe représentant un hôte de la distribution. Elle permet de définir la politique d'envoi du message.
- la méthode `receive` définit le décodage à effectuer lors de la réception d'une réplication envoyée avec les propriétés de communication que la classe décrit. Elle doit donc effectuer les traitements inverses de la méthode `processMessage`, pour tenter de reconstruire la représentation de l'état dont elle a reçu la réplication, puis déléguer le traitement du résultat à la classe qui effectuera la mise à jour de l'état.

Lorsqu'un état est répliqué vers un hôte donné, le *framework* appelle tout d'abord la méthode `processMessage`. Il préfixe ensuite chaque chaîne de caractères de la liste obtenue par l'identifiant de la propriété de communication utilisée, puis appelle la méthode `send`.

Lorsque l'application reçoit un message, le *framework* utilise le préfixe pour en adresser le traitement à la propriété de communication adéquate et appelle sa méthode `receive`.

La méthode `send` est celle qui doit définir la politique d'envoi du message proprement dit : soit elle écrit directement les messages sur les canaux de communication définis par la classe représentant l'hôte auquel envoyer le message, soit elle lui délègue cet envoi, par exemple pour appliquer une politique de quorum de temps ou de quantité d'informations à atteindre avant d'effectuer l'envoi.

Il sera donc parfois nécessaire, en plus de définir une nouvelle classe de description des propriétés de communication, de définir de nouvelles briques de base étendant une des classes qui permet de représenter les différents hôtes de l'application.

Voici un exemple très simple de deux classes permettant de définir une réplication transmise de manière non fiable et non ordonnée, en utilisant un canal UDP.

Un hôte de l'application qui dispose de sockets TCP et UDP

```
public class TCPUDPRecipient extends TCPRecipient {

    private DatagramSocket socket ;
    private InetAddress inetAddress ;
    private int port ;
    public TCPUDPRecipient(long id, BufferedWriter output,
                           BufferedReader input) {
        super(id,output, input) ;
    }
    public synchronized DatagramSocket getSocket() {
        return socket;
    }
    public synchronized InetAddress getInetAddress() {
        return inetAddress;
    }
    public synchronized int getPort() {
        return port;
    }
    public synchronized void setInetAddress(InetAddress inetAddress) {
        this.inetAddress = inetAddress;
    }
```

```
    public synchronized void setPort(int portSender) {
        this.port = portSender;
    }
    public synchronized void setSocket(DatagramSocket socket) {
        this.socket = socket;
    }
}
```

Une propriété de communication utilisant le protocole UDP

```
public class UDPProtocol extends CommunicationProperties {

    public UDPProtocol(String key) {
        super(key);
    }

    protected NetworkMessage processMessage(String msg) {
        NetworkMessage message = new NetworkMessage() ;
        message.messageItems.add(msg) ;
        return message;
    }
    protected void sendMessage(NetworkMessage msg, Recipient recipient) {
        TCPUDPRecipient tcpudpRecipient = (TCPUDPRecipient) recipient ;
        DatagramSocket output = tcpudpRecipient.getSocket() ;
        byte[] message = ((String)msg.messageItems.get(0)).getBytes() ;
        DatagramPacket packet = new DatagramPacket(message,message.length,
              tcpudpRecipient.getInetAddress(),tcpudpRecipient.getPort()) ;
        try {
            output.send(packet) ;
        } catch (IOException e) {
            System.err.println(e.getMessage()) ;
            recipient.garbage() ;
        }
    }
    public void receive(String message, Recipient recipient) {
        ProtocolExtraction.extractProtocol(message,recipient) ;
    }
}
```

6.3 L'ordonnanceur équitable des réflexes

La figure 6.1 décrit les relations entre les classes qui gèrent le calcul local de l'application, l'ordonnancement des réflexes. Nous allons commencer par

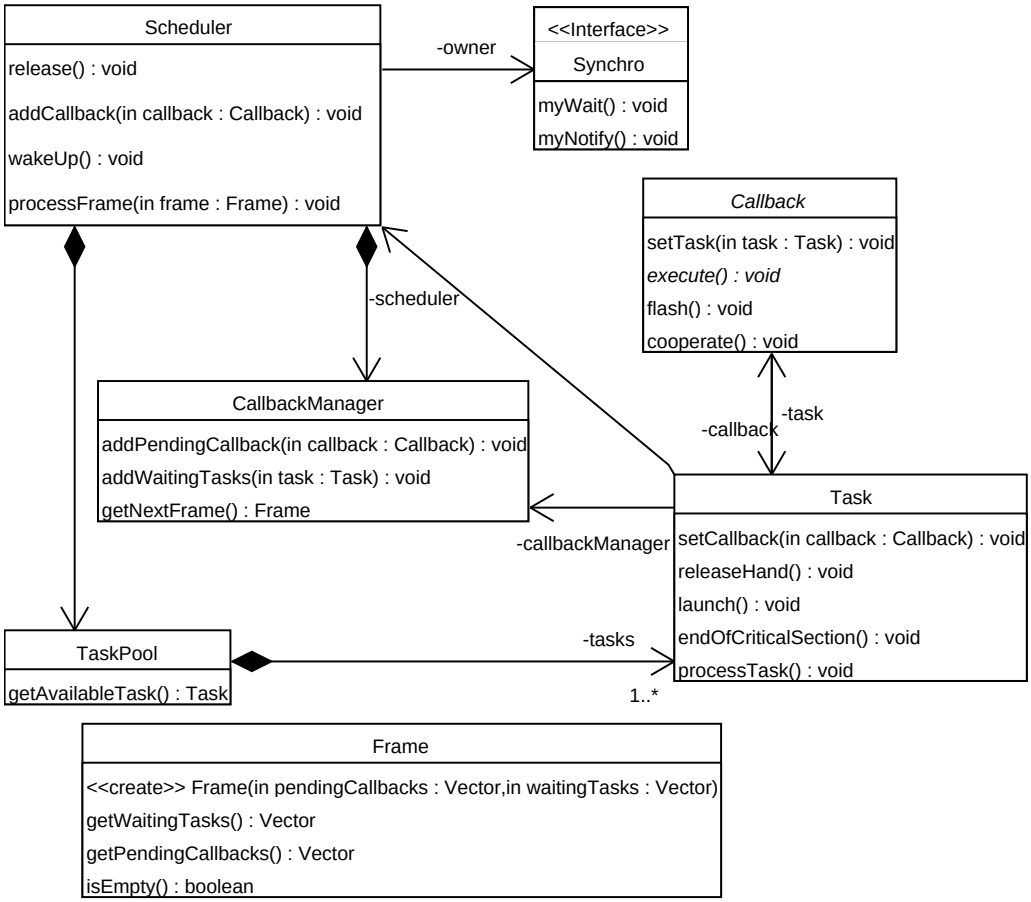


FIG. 6.1 – Ordonnancement des réflexes, diagramme de classes

décrire les responsabilités de ces différentes classes.

L'interface `Synchro` est implémentée par la classe principale (celle qui contient le programme principal) de l'application. Elle permet de synchroniser le démarrage de l'ordonnanceur avec celui de l'application, afin qu'aucune modification n'intervienne tant que celui-ci n'est pas prêt.

La classe `Callback` représente un réflexe défini par l'utilisateur, lors de l'assemblage d'un ordonnançable. Sa méthode `execute` permet à l'ordonnanceur de lancer l'exécution du réflexe. C'est dans cette classe que sont définies les méthodes correspondant aux instructions utilisateur dont nous avons défini la sémantique au chapitre précédent (voir section 5.4.1, page 141).

La classe `Scheduler` doit être instanciée par l'application principale. Elle crée les classes `TaskPool` et `CallbackManager`. C'est cette classe, qui étend la classe Java *Thread*, qui ordonnance l'exécution des réflexes les uns après les autres. L'instance de cette classe, créée au démarrage de l'application, fait également office d'interface entre le mécanisme d'ordonnancement représenté par les classes qui suivent et le reste de l'application.

La classe TaskPool est responsable de la gestion des *threads* Java qui serviront à exécuter les réflexes des ordonnançables en parallèle. Un certain nombre de *threads* Java sont créés dès le démarrage.

La classe `CallbackManager` est responsable de la gestion des réflexes en attente d'exécution, après une interruption par la méthode *cooperate* de la classe `Callback` de la manière décrite par la règle $COOPERATE$ de la sémantique du calcul local (voir section 5.4.1, page 141), ou juste après leur déclenchement. Il correspond à l'ensemble S du prédicat utilisé pour décrire l'état de l'application dans la sémantique du calcul local.

La classe `Task` représente un *thread* Java, auquel l'ordonnanceur peut affecter un réflexe à exécuter.

Enfin, la classe `Frame` est une liste d'objets **Task** en attente et de réflexes activés mais pas encore exécutés, et est utilisée pour construire un tour d'exécution des réflexes qui s'exécutent en parallèle.

La figure 6.2 décrit l'ajout d'un réflexe à la file d'exécution lors de l'activation d'un ordonnançable. Cet ajout est lié au déclenchement d'un ordonnançable, suite à la mise à jour d'un état de la manière décrite par la règle $UPDATE(s)$ de la sémantique du calcul local (voir section 5.4.1, page 141) ou conséquence d'une réplication de la manière décrite par la règle $UPDATE\ \ REP(s)$ de la sémantique du calcul distribué (voir section 5.4.2, page 148) . Si l'ordonnanceur est en sommeil faute de réflexes à exécuter, il est alors réveillé.

La figure 6.3 décrit l'exécution d'un tour de l'ordonnanceur. Lorsque l'or-

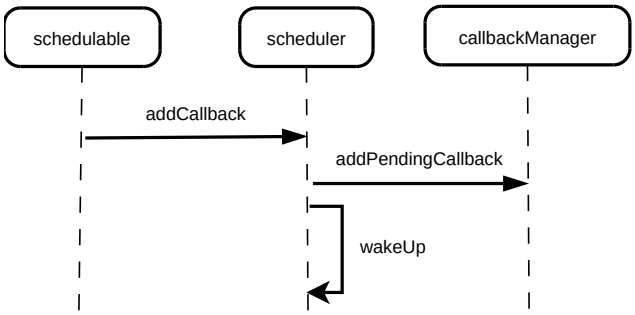


FIG. 6.2 – Ajout d'un réflexe à la file de l'ordonnanceur lors du déclenchement de l'ordonnançable

donnanceur est réveillé, il effectue en boucle le traitement décrit.

Il construit l'objet `frame` correspondant au prochain tour d'exécution, et commence par exécuter les premiers blocs d'instructions de chaque réflexe en attente de démarrage, qui ont été ajoutés à la file d'attente lors du tour précédent selon la figure 6.2. Pour ce faire, il affecte chaque réflexe dans un objet `task` qui n'est pas déjà affecté à l'exécution d'un autre réflexe.

Puis il passe à l'exécution des blocs suivants des réflexes ayant appelé l'opération *cooperate* au tour précédent.

Si l'objet `frame` correspondant au tour suivant est vide, le `scheduler` se met en sommeil, et attend d'être réveillé par l'ajout d'un réflexe à exécuter.

6.4 Dynamique de l'application

La figure 6.4 représente la manière dont sont implémentés les différents composants qui constituent l'application. Nous allons passer en revue ces composants, et la manière dont ils interagissent. La classe abstraite `State` est la classe permettant de définir les états de l'application (définition 4.5.1, page 113), l'extension `NetState` de cette classe permet de définir des états abonnés au réseau, qui pourront être mis à jour par des hôtes distants.

La classe `DuplicatedStatesSet` est une classe rassemblant des états duplicata (voir section 5.3.1.2, page138). Un objet de cette classe n'est pas un état, mais une collection d'états se comportant de manière similaire et

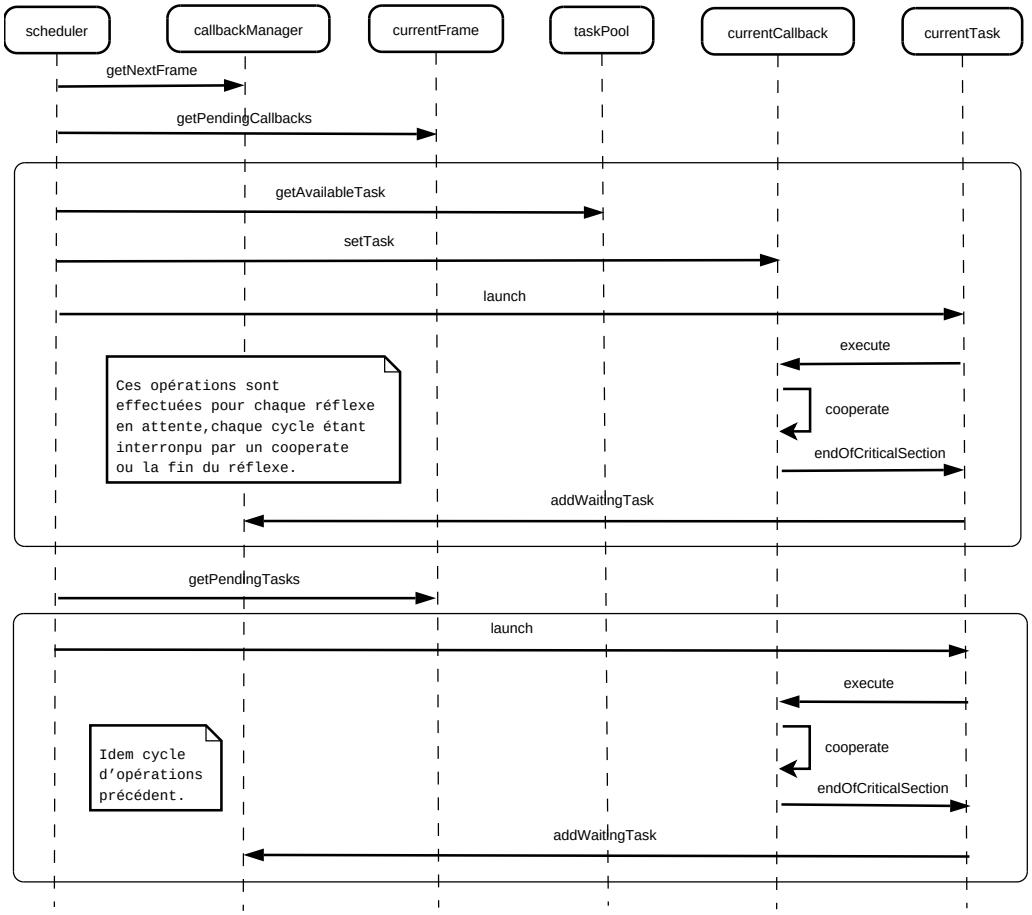


FIG. 6.3 – Description d'un tour d'exécution

ayant le même père dans la hiérarchie en arbre des états. La classe `State` et la classe `DuplicatedStatesSet` ont donc néanmoins un certain nombre de méthodes et d'attributs en commun, dû au fait qu'ils sont tous deux des composants des états. Comme l'héritage multiple n'est pas possible en Java, nous avons utilisé pour unifier ces deux types l'interface `StateComponent`, et factorisé leurs traitement commun à la classe `ConcreteStateComponent` à qui les classes `State` et `DuplicatedStateSet` délèguent ces traitements.

La classe abstraite `Schedulable` est la classe qui représente les ordonnançables (définition 5.2.1, page 134) et ses extensions `Behavior` et `Replication` permettent de définir respectivement les ordonnançables comportementaux

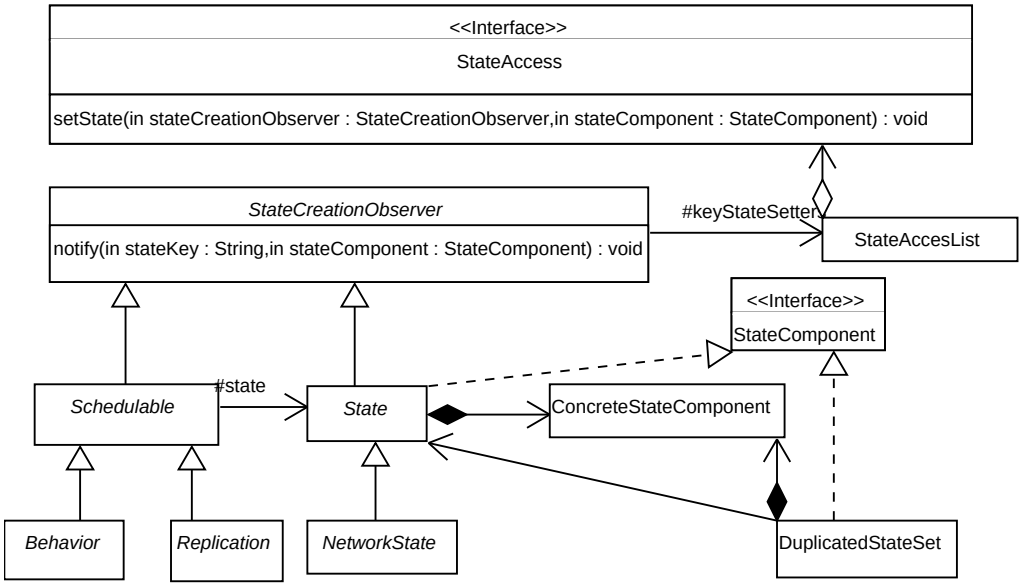


FIG. 6.4 – Architecture des composants, diagramme de classes

(les ordonnançables sans réplication associée) et les modèles de réplication. Les objets de type `Schedulable` correspondent au patron de conception classique *Observateur* [46], et sont notifiés lorsqu'ils sont instanciés, et qu'un des états observés (définition 5.3.3, page 140) est modifié. La figure 6.5 représente l'enchaînement d'appels qui mène à l'ajout d'un réflexe à la file d'attente de l'ordonnanceur lorsqu'un état est modifié et que certains de ses ordonnançables sont déclenchés. Cette séquence correspond à la règle $UPDATE(s)$ de la sémantique du calcul local que nous avons décrit au chapitre précédent (voir section 5.4.1, page 141). Elle sera alors suivie des séquences que nous avons déjà présentées pour décrire le fonctionnement de l'ordonnanceur (figure 6.2, page 170, et figure 6.3, page 171). La figure 6.6 décrit le mécanisme d'envoi d'une mise à jour lorsqu'un modèle de réplication notifié est activé et que sa temporalité est vérifiée. Cette séquence correspond à la production par le processus d'un événement $R(s, S)$, tel que décrit dans la sémantique du calcul distribué (voir section 5.4.2, page 148) par les règles $REPLICATE\ UPDATE(s)$.

La classe abstraite `StateCreationObserver`, étendue par les classes d'ordonnançables et les classes d'états, correspond également au patron de concep-

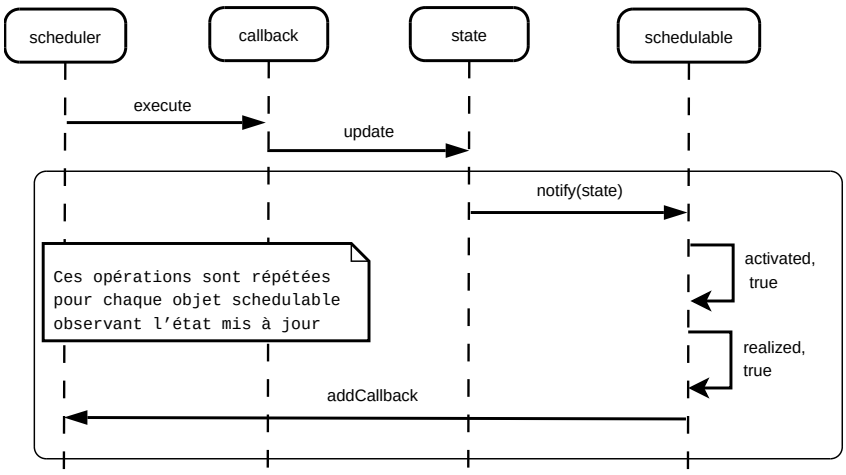


FIG. 6.5 – Mise à jour d'un état par un réflexe, diagramme de séquence

tion classique *Observateur*, et est notifié par l'application chaque fois qu'un état pour lequel il a exprimé un intérêt est créé ou détruit. Elle contient les listes d'associations définies dans les fabriques des composants, et les utilise pour affecter les états, les états duplicata, et les ensembles d'états duplicata dont la création lui a été notifiée dans les attributs adéquats. La figure 6.7 décrit ce mécanisme d'affectation aux composants de l'application qui observent les créations. Cette séquence permet notamment d'implémenter les comportements des règles $CREATE(s)$ de la sémantique du calcul local (voir section 5.4.1, page 141) qui illustrent l'activation d'un ordonnançable instancié.

L'application crée les états et les ensembles d'états duplicata au travers d'un objet `mapStorage` qui est la carte de l'application. Nous étudierons plus en détail cette classe dans la partie suivante. Pour les états, ces notifications interviennent donc lors de la création ou de la destruction de ses sous-états. Pour les ordonnançables, elles interviennent lors de la création des états utiles dans le calcul des conditions et des réflexes.

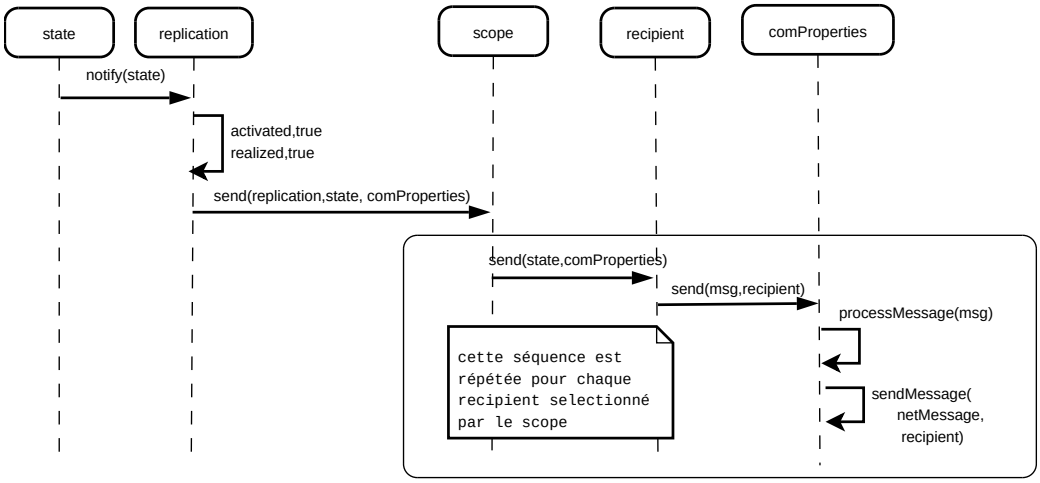


FIG. 6.6 – Réplication d'un état, diagramme de séquence

6.5 La carte de l'application

La carte de l'application `MapStorage` (figure 6.8) rassemble tous les composants de l'application ainsi que l'état local du jeu. Une instance de cette classe est créée lors du démarrage. Elle contient la mémoire partagée de l'application, et tous les objets la composant, ainsi que ceux qui permettront de créer les composants non encore instanciés. Elle est responsable de ces créations et destructions.

Lors du démarrage de l'application, chaque fabrique définie par l'utilisateur pour la création des états et des ordonnançables est utilisée pour créer une ou plusieurs associations avec les textes de configuration qui décrivent le composant à créer. Ces associations sont enregistrées dans des objets de type `*Creator` pour chaque composant. Ainsi, les objets fabriques peuvent être utilisés pour la création de différentes instances de composants.

Les objets de type `*Creator` peuvent également être créés dynamiquement, par exemple lors de la réception d'une demande de création d'état par un hôte distant, tel que décrit dans la sémantique du calcul distribué (voir section 5.4.2, page 148) par la règle $CREATE\ REP(s)$.

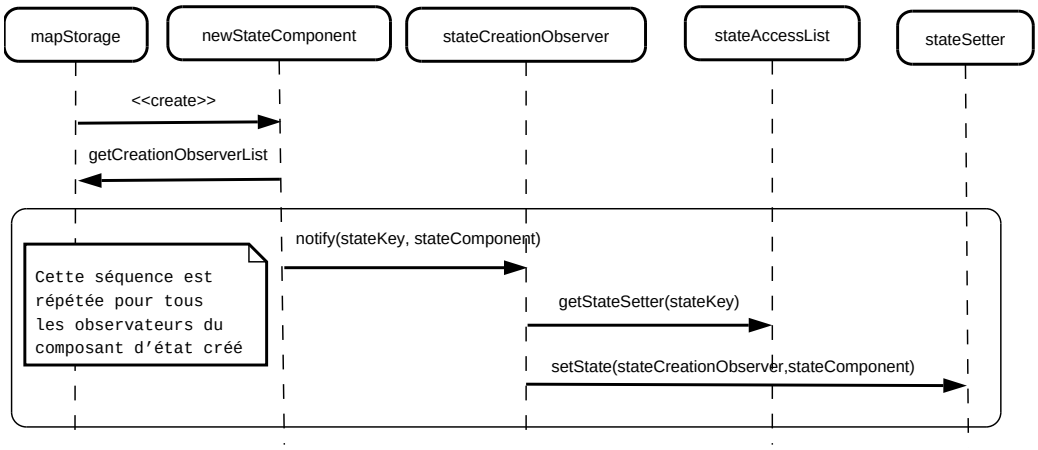


FIG. 6.7 – Notification des observateurs lors de la création d'un composant d'état

6.6 Un exemple simple

Nous proposons un exemple très simple pour illustrer l'utilisation du *Fill-In-The-Gaps Toolkit*.

Il s'agit d'une application clients-serveur, dont les hôtes dialoguent à l'aide de connexions TCP-IP et de pseudo-connexions UDP-IP.

Les participants peuvent déplacer leurs avatars, chacun percevant les déplacements des autres.

Les utilisateurs peuvent également communiquer entre eux à l'aide d'une interface textuelle, de deux manières différentes. Lorsqu'un participant *murmure*, en faisant précéder son message par la commande `/murmure`, seuls les participants dont les avatars sont à proximité peuvent percevoir ce qu'il transmet. Sinon, tout le monde reçoit ses messages.

La figure 6.9 montre l'interface cliente de cette application. L'avatar de l'utilisateur est représenté en rose, et le carré bleu qui l'entoure représente la zone à l'intérieur de laquelle les autres participants peuvent recevoir ses *murmures*. Nous allons décrire comment cette application a été construite en utilisant le *Fill-In-The-Gaps Toolkit*. Nous nous concentrerons principalement sur la description du serveur, l'architecture des états de l'application cliente étant très proche, et ses modèles de réplication plus simples.

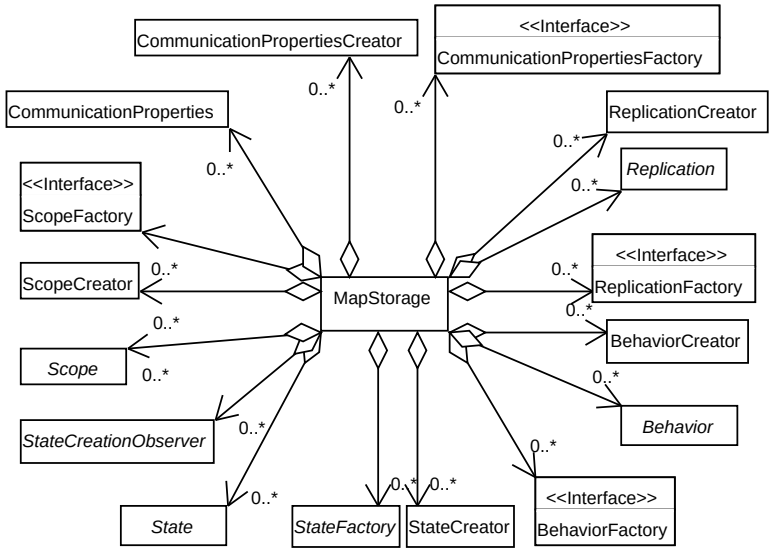


FIG. 6.8 – Carte de l'application, diagramme de classes

Cet exemple nous permet également de montrer une approche canonique pour la réalisation d'une architecture clients-serveur à l'aide du *framework*.

6.6.1 Les états du jeu

Un état racine `gameState` représente l'état du jeu, et rassemble les autres états. La figure 6.10 décrit son architecture sur le serveur, que nous allons commenter.

L'ensemble d'états duplicata `clients` regroupe les états représentant les clients de l'application. La classe `Client` représentant un client de l'application étend la classe `NetworkedState` du *framework*, ce qui signifie que chaque état duplicata peut recevoir des mises à jour par des hôtes distants de l'application.

L'état `lastHostConnected` contient l'identifiant de l'état duplicata correspondant au dernier client qui s'est connecté au serveur. Cet état étend la classe `State` du *framework*, et n'est donc pas susceptible d'être modifié par un hôte distant. En fait, c'est un état local, qui ne sera pas non plus sujet à réplication.

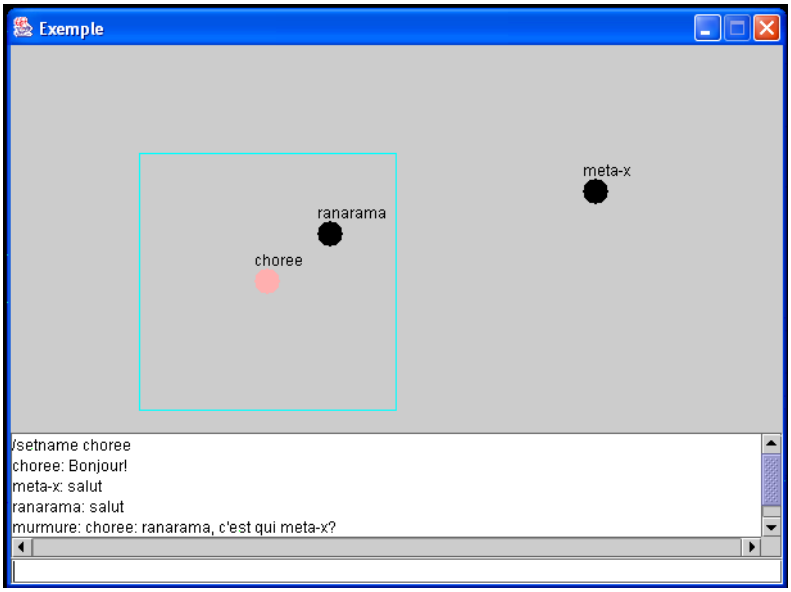


FIG. 6.9 – Le client d'une application de chat

L'état `lastMessage` contient une chaîne de caractères représentant le dernier message envoyé par un client par son interface de communication en texte. Cet état étend donc la classe `NetworkedState` du *framework*, car il est répliqué depuis les clients.

L'état `newMessage` contient une chaîne de caractères représentant le prochain message à envoyer aux clients pour leur interface de communication en texte. Cet état étend la classe `State` du *framework*.

L'état `murmureDistance` contient la valeur utilisée pour décider si l'avatar d'un client est en mesure d'entendre un *murmure* envoyé par un autre client.

La classe `LastHostConnected` permettant de décrire l'état `lastHostConnected` ainsi que sa fabrique font partie de la bibliothèque des blocs de base.

La classe `Client` permettant de décrire un état duplicata de l'ensemble `clients` est également utilisée pour la description de l'application cliente. Il en est de même pour les classes de description `MessageState` et `MessageNetworkedState` utilisées pour les états `lastMessage` et `newMessage`, ainsi que pour les fabriques associées à ces deux classes.

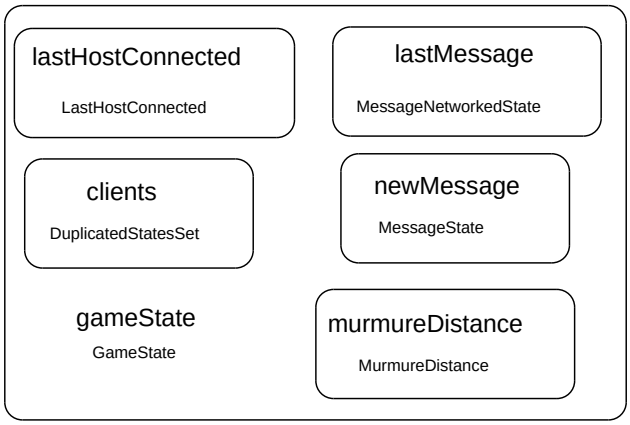


FIG. 6.10 – L'état `gameState` sur le serveur

Le lecteur curieux ou insatisfait peut trouver en annexe le code des classes de description de ces états et les fabriques associées, que nous voulons à terme permettre de générer automatiquement à partir d'un outil d'intégration construit autour du *Fill-In-The-Gaps Toolkit*.

6.6.2 Les hôtes de l'application

Pour décrire l'application cliente et serveur, nous nous sommes appuyés sur une classe de la bibliothèque des blocs de base, qui permet de représenter un hôte de l'application qui comporte un canal TCP-IP, une adresse et un numéro de port auquel on peut envoyer des datagrammes UDP-IP. Cette classe étend la classe `Recipient` du *framework*, qui contient déjà les primitives de gestion d'un hôte de la distribution.

6.6.2.1 La classe représentant un hôte de l'application

Une première classe permet de représenter un hôte avec lequel on est en communication TCP-IP, et utilise les primitives de haut niveau de la bibliothèque standard de Java :

```
public class TCPRecipient extends Recipient{
```

```
    private BufferedWriter output ;
    private BufferedReader input ;
    public TCPRecipient(final long id, final String name ,
        BufferedWriter output , BufferedReader input) {
        super(id, name) ;
        this.output = output ;
        this.input = input ;
    }
    public synchronized BufferedWriter getOutput() {
        return output;
    }

    public synchronized BufferedReader getInput() {
        return input;
    }
}
```

Nous avons étendu cette classe, pour fournir la description d'un hôte qui dispose des informations nécessaires pour l'envoi de datagrammes UDP-IP à une adresse et un numéro de port donnés de l'hôte. Cette classe a une référence vers la *socket* locale à utiliser pour envoyer le datagramme.

```
public class TCPUDPRecipient extends TCPRecipient {

    private DatagramSocket socket ;
    private InetAddress inetAddress ;
    private int port ;
    public TCPUDPRecipient(long id, String name,
        BufferedWriter output,BufferedReader input,
        InetAddress inetAddress) {
        super(id,name,output, input) ;
        this.inetAddress = inetAddress ;
    }
    public synchronized DatagramSocket getSocket() {
        return socket;
    }
    public synchronized InetAddress getInetAddress() {
        return inetAddress;
    }
    public synchronized int getPort() {
```

```
        return port;
    }
    public synchronized void setPort(int portSender) {
        this.port = portSender;
    }
    public synchronized void setSocket(DatagramSocket socket) {
        this.socket = socket;
    }
}
```

6.6.2.2 Un client et un serveur

Nous utilisons ensuite des classes abstraites de la bibliothèque des blocs de base, donnant respectivement :

- une description d'un serveur TCP-IP concurrent ;
- une description d'un client TCP-IP ;
- un service de réception de messages UDP-IP lié à une *socket* de la classe Java `DatagramSocket`.

Nous utilisons ces trois éléments pour décrire les classes principales de l'application : le logiciel client et le logiciel serveur.

Ces trois éléments, lorsqu'ils reçoivent des messages, appellent le service de traitement des mises à jour interne au *framework*. Ce dernier route le message à la *propriété de communication* utilisée pour l'envoi par l'hôte émetteur, qui traite alors la mise à jour.

A partir de ces éléments de la bibliothèque, nous avons à décrire ce qui se passe lorsqu'un client se connecte au serveur. Cette description est spécifique à l'architecture choisie, mais peut être vue comme la manière canonique d'organiser l'application distribuée en utilisant le *Fill-In-The-Gaps Toolkit*.

- La méthode abstraite `connected(Socket s)` de la classe définissant un serveur TCP-IP concurrent, et définie par notre application serveur, est appelée.
- Cette méthode crée un état duplicata correspondant au nouveau client connecté, et crée un objet de la classe `TCPUDPRecipient` dont le nom correspond à l'identifiant du nouvel état duplicata. Ceci permettra ultérieurement aux *portées* que nous allons définir de faire le tri entre les différents hôtes à qui répliquer une mise à jour d'état. Cette pratique

canonique permet de construire des *portées* sélectionnant les hôtes à qui envoyer une réplication.
- S'ensuit un dialogue entre le client connecté et le serveur, afin de se communiquer les numéros de ports utilisés pour leurs services de communication UDP-IP. Dans notre implémentation, un seul port UDP serveur reçoit donc les mises à jour de tous les clients par ce protocole.
- Finalement, le serveur ajoute à l'objet décrivant l'hôte la clé de l'état duplicata représentant le client connecté, pour lui donner l'autorisation de modifier cet état par réplication, et met à jour l'état `lastHostConnected` avec l'identifiant de cet hôte.

6.6.3 Les propriétés de communication

La bibliothèque des blocs de base comprend, en plus des deux classes `TCPRecipient` et `TCPUDPRecipient`, deux *propriétés de communication* très simples et leurs fabriques, reposant entièrement sur les protocoles TCP-IP et UDP-IP et n'effectuant aucune transformation des messages de mise à jour. Ces classes utilisent les classes `TCPRecipient` et `TCPUDPRecipient` définies précédemment :

```
public class TCPProtocol extends CommunicationProperties {

    public TCPProtocol(String key) {
        super(key);
    }
    protected NetworkMessage processMessage(String msg) {
        NetworkMessage message = new NetworkMessage() ;
        message.messageItems.add(msg) ;
        return message;
    }
    protected void sendMessage(NetworkMessage msg, Recipient recipient) {
        BufferedWriter output = ((TCPRecipient) recipient).getOutput() ;
        try{
            output.write((String)msg.messageItems.get(0)) ;
            output.newLine() ;
            output.flush() ;
        }
        catch (IOException e) {
```

```
            recipient.garbage() ;
        }
    }
    public void receive(String message, Recipient recipient) {
        ProtocolExtraction.extractProtocol(message,recipient) ;
    }
}

public class UDPProtocol extends CommunicationProperties {

    public UDPProtocol(String key) {
        super(key);
    }
    protected NetworkMessage processMessage(String msg) {
        NetworkMessage message = new NetworkMessage() ;
        message.messageItems.add(msg) ;
        return message;
    }
    protected void sendMessage(NetworkMessage msg, Recipient recipient) {
        TCPUDPRecipient tcpudpRecipient = (TCPUDPRecipient) recipient ;
        DatagramSocket output = tcpudpRecipient.getSocket() ;
        byte[] message = ((String)msg.messageItems.get(0)).getBytes() ;
        DatagramPacket packet = new DatagramPacket(message,message.length,
              tcpudpRecipient.getInetAddress(),tcpudpRecipient.getPort()) ;
        try {
            output.send(packet) ;
        } catch (IOException e) {
            System.err.println(e.getMessage()) ;
            recipient.garbage() ;
        }
    }
    public void receive(String message, Recipient recipient) {
        ProtocolExtraction.extractProtocol(message,recipient) ;
    }
}
```

Les modèles de réplication de l'application utiliseront des propriétés de communication qui seront des instances de ces classes.

La propriété de communication tcpProtocol : c'est une instance de la classe `TCPProtocol`.

La propriété de communication udpProtocol : c'est une instance de la classe `UDPProtocol`.

6.6.4 Les portées

Les modèles de réplication de l'application utilisent plusieurs portées, dont certaines sont décrites dans la bibliothèque des blocs de base.

La portée allScope : sélectionne l'ensemble des clients de l'application. Cette portée fait partie de la bibliothèque des blocs de base.

La portée allButMeScope : sélectionne l'ensemble des hôtes, sauf celui à qui appartient l'état modifié. Elle effectue la sélection en comparant l'identifiant de l'état modifié, qui comprend la liste des identifiants de ses ancêtres dans l'arbre de ses états, et le nom de chaque hôte. Cette portée fait partie de la bibliothèque des blocs de base.

La portée hostCreatedScope : sélectionne l'hôte correspondant à l'identifiant dans l'état `lastHostCreated`. Elle est intimement liée au modèle de réplication qui l'utilise, puisque c'est sa copie de l'état `lastHostCreated` qu'il utilise dans ses calculs.

La portée murmureScope : sélectionne l'ensemble des hôtes dont les états duplicata `clients` correspondant sont dans la zone de murmure dont la taille est définie par l'état `murmureDistance`, par rapport à la position de l'état duplicata correspondant à l'hôte qui a envoyé le *murmure*. De même que la portée précédente, cette portée est intimement liée au modèle de réplication qui l'utilise. Les classes de description de ces portées font partie de la bibliothèque des blocs de base, excepté pour la portée `murmureScope`, qui est liée à un modèle de réplication spécifique à l'application. Le lecteur pourra trouver en annexe les classes de description pour cette portée.

Nous donnons en exemple le code de la classe qui sert à instancier la portée `hostCreatedScope`, qui utilise un état utile du modèle de réplication auquel il est attaché :

```
public class HostConnectedScope extends Scope {

    public HostConnectedScope(String key) {
        super(key);
    }
    public void send(Replication replication, State state,
        CommunicationProperties protocol) {
        HostConnectedReplication hostConnectedReplication =
            (HostConnectedReplication)replication ;
        Recipient recipient = hosts.findRecipient(
          hostConnectedReplication.getLastHostConnected().getHostIdentifier()) ;
        recipient.send(state, protocol) ;
    }
}
```

6.6.5 Les ordonnançables, côté serveur

Maintenant que nous avons présenté les états et les briques de base nécessaires, nous pouvons définir les ordonnançables qui décrivent le comportement de l'application.

On définit cinq ordonnançables pour la description des comportements et les réplications des états du serveur. Pour chaque ordonnançable, nous donnerons le nom de la classe qu'il instancie, afin que le lecteur curieux puisse se référer au code Java de cette classe, en annexe.

Le modèle de réplication forwardReplication : c'est une instance de la classe `StateChangedReplication` de la bibliothèque des blocs de base.

- Il est **attaché** à chaque état duplicata représentant un client.
- Il a comme **état observé** l'état duplicata auquel il est attaché.
- Il ne définit aucun **état utile**.
- Sa **condition de réalisation** (temporalité) renvoie toujours `true`.
- Il ne définit aucun **réflexe**.
- Sa **portée** est `allButMeScope`.

- Ses **propriétés de communication** sont `udpProtocol`.

Ce modèle sera donc déclenché lors du changement de chaque état duplicata correspondant à un client de l'application, pour propager ce changement aux autres hôtes clients. Il permettra notamment de propager les déplacement des avatars.

Le modèle de réplication hostCreatedReplication : c'est une instance de la classe `HostCreatedReplicationq` de la bibliothèque des blocs de base.

- Il est **attaché** à chaque état duplicata représentant un client.
- Il a comme **état observé** l'état `lastHostConnected`.
- Il a comme **état utile** l'état `lastHostConnected`.
- Sa **condition de réalisation** (temporalité) renvoie toujours `true`.
- Il ne définit aucun **réflexe**.
- Sa **portée** est `hostCreatedScope`.
- Ses **propriétés de communication** sont `tcpProtocol`.

Ce modèle de réplication sera déclenché à chaque mise à jour de l'état `lastHostConnected`, et utilisera la valeur de cet état pour propager à l'hôte correspondant les états duplicata représentant les autres hôtes de l'application. Il permettra de provoquer la création de l'état correspondant au dernier client connecté aux autres clients de l'application.

L'ordonnançable sans réplication newMessageBehavior : c'est une instance de la classe `StateUpdateBehavior`.

- Il est **attaché** à l'état `gameState`.
- Il a comme **état observé** l'état `lastMessage`.
- Il a comme **états utiles** les états `lastMessage` et `newMessage`.
- Sa **condition de réalisation** (temporalité) renvoie toujours `true`.
- Son **réflexe** copie le contenu de l'état `lastMessage` dans l'état `newMessage`.

Cet ordonnançable sera déclenché chaque fois que l'état `lastMessage` est modifié, et modifiera l'état `newMessage`.

Le modèle de réplication chatMessage : c'est une instance de la classe `ChatMessageReplication`.

- Il est **attaché** à l'état `newMessage`.
- Il a comme **état observé** l'état `newMessage`.
- Il a comme **état utile** l'état `newMessage`.

- Sa **condition de réalisation** (temporalité) renvoie `true` si et seulement si le texte contenu dans newMessage ne commence pas par la commande *murmure*.
- Il ne définit aucun **réflexe**.
- Sa **portée** est `allScope`.
- Ses **propriétés de communication** sont `tcpProtocol`.

Ce modèle de réplication sera déclenché chaque fois que `newMessage` sera modifié et que la chaîne de caractères qu'il contient ne commencera pas par la commande de *murmure*. Il répliquera alors l'état `newMessage` vers tous les clients de l'application.

Le modèle de réplication murmureMessage : c'est une instance de la classe `MurmureMessageReplication`.

- Il est **attaché** à l'état `newMessage`.
- Il a comme **état observé** l'état `newMessage`.
- Il a comme **état utile** l'état `newMessage` et l'état `murmureDistance`.
- Sa **condition de réalisation** (temporalité) renvoie `true` si et seulement si le texte contenu dans `newMessage` commence par la commande *murmure*.
- Il ne définit aucun **réflexe**.
- Sa **portée** est `murmureScope`.
- Ses **propriétés de communication** sont `tcpProtocol`.

Ce modèle de réplication sera déclenché chaque fois que `newMessage` sera modifié et que la chaîne de caractères qu'il contient commencera par la commande de *murmure*. Il répliquera alors l'état `newMessage` vers tous les clients à portée du murmure en utilisant l'état `murmureDistance` pour sélectionner les hôtes correspondant.

Le lecteur curieux ou insatisfait peut trouver en annexe le code correspondant à la description de ces ordonnançables et leurs fabriques, ainsi que les textes de configuration qui définissent chacun de ces modèles de réplication.

6.6.6 Synthèse

Bien que cet exemple soit très simple, et n'illustre pas l'utilisation des instructions *flash* et *cooperate* dans la définition des réflexes utilisateur, il

suffit à montrer comment nous avons rempli les objectifs que nous nous étions fixés.

La gestion des données représentant l'état de l'application, organisées en une hiérarchie d'états, est totalement découplée de la manière dont ces données sont synchronisées sur les différents hôtes, définie à l'aide des modèles de réplication. Ce découplage facilite la mise au point des interactions, car il rend leur définition indépendante de la modélisation de l'application en terme d'objets du monde virtuel.

Ce découplage entre les données et leurs traitements nous permet de modifier simplement les propriétés de réplication d'un état, sans avoir à toucher à l'organisation des données déjà définie : par exemple, si on veut augmenter la distance de murmure, il suffit de modifier la valeur de l'état **distanceMurmure**. On peut même ajouter un ordonnançable qui modifie cette distance en fonction, par exemple, du nombre de clients connectés. De même, il est aisé de remplacer une propriété de communication ou une portée par une autre dans la définition d'un modèle de réplication. Il est également aisé d'ajouter un modèle de réplication à un état : par exemple, si on décide de vouloir enregistrer la position de chaque joueur lorsqu'il se déconnecte, on peut définir un nouveau modèle de réplication pour communiquer les positions des avatars de manière fiable. Ces modifications n'auront aucun impact sur la définition de l'état répliqué, seul son texte de configuration sera à modifier légèrement.

Cette approche, couplée avec une méthode de développement basée sur le raffinement de prototypes et des tests systématiques (en utilisant des outils de simulation de réseau par exemple), permet de construire une application autour de la définition et de la mise au point des interactions, en les adaptant très précisément au *game-play* désiré.

Cet exemple montre également comment utiliser le *framework* et la bibliothèque des blocs de base pour mettre en place une architecture donnée. Il peut être vu comme un cas d'école, à utiliser comme modèle pour réaliser d'autres architectures de distribution.

Nous sommes toutefois conscients que pour l'instant, il est délicat d'utiliser notre solution, tant qu'il n'existe pas d'outil permettant de générer le code lourd et répétitif décrivant états et ordonnançable.

Chapitre 7

Perspectives

7.1 Synthèse des travaux réalisés

Nous avons proposé dans le cadre de cette thèse une nouvelle approche dédiée à la réalisation de jeux massivement multi-joueurs, permettant de rationaliser et de sécuriser le développement de jeux innovants en ce qui concerne la manière dont le joueur interagit avec le monde virtuel dans lequel le jeu se déroule.

Notre proposition consiste à utiliser des méthodes de génie logiciel appropriées, basées sur une démarche de prototypage et de raffinement, en utilisant un environnement de développement dédié.

Nous avons défini et prototypé un framework ouvert et modulaire pour garantir la généricité de la solution, et montré comment il peut s'intégrer dans un environnement de développement.

Ce modèle original est centré sur la définition des interactions à l'aide d'une description très fine de la manière dont les données de l'application sont répliquées sur les différents hôtes de l'application distribuée. Il adopte un modèle simple de programmation concurrente pour la description du comportement de l'application, basé sur un ordonnancement coopératif et équitable des tâches en cours d'exécution.

Plus général que les modèles orientés objet et orientés agents, il dissocie

la manière dont sont organisées les données de leurs comportement. Il permet néanmoins de concevoir une application selon ces paradigmes, en organisant de manière adéquate les données et les traitements.

Ce découplage entre les données et leur traitement est complété par une approche transverse, proche des modèles de meta-programmation tels que la programmation par aspects, où différentes manières d'effectuer un traitement peuvent être considérées lors de la conception de l'application.

7.2 Implémentation réaliste

Nous voulons dans un futur proche réaliser une implantation plus réaliste du modèle, en prenant en compte les autres aspects d'un jeu massivement multi-joueurs, dont les plus importants sont donnés ici.

Nous voulons ajouter au prototype un mécanisme permettant aux hôtes de l'application de s'échanger ou de mettre à jour dynamiquement de nouveaux comportements (les ordonnançables). Cette amélioration ne modifiera pas le modèle, mais seulement le prototype, puisque nous envisageons alors d'ajouter les comportements à la liste des types de base définissant les données propres d'un état. Un état pourra ainsi être conçu pour définir un moyen d'obtenir un mécanisme de *code mobile*.

Nous aimerions également réaliser une implantation plus performante que celle qui existe actuellement, et qui est basée sur l'ordonnancement des *threads* Java afin de réaliser un autre ordonnanceur. L'implantation actuelle utilise un *pool* de *threads* Java qui sont tous créés au lancement de l'application : la création d'un thread est très coûteuse en Java et nous devions éviter d'utiliser trop souvent cette opération pour garder un prototype un tant soit peu réaliste. Lorsqu'un réflexe est déclenché, son exécution est attribuée à un *thread* donné du *pool*. Le nombre de *threads* à créer au démarrage, ainsi que son augmentation ou diminution dynamique selon les besoins de l'application est un problème complexe. Il existe des patrons de conception répondant à cette famille de problèmes et nous comptons les étudier et expérimenter afin de trouver une solution souple et efficace. Nous n'envisageons pas forcément d'abandonner totalement le langage Java, qui peut rester le language d'implantation du reste du *framework* et de la bibliothèque, mais il est possible

que d'autres langages soient plus adéquats pour la réalisation de l'ordonnanceur lui-même, et nous comptons faire quelques expériences à ce sujet.

De même, nous comptons dans un futur très proche étendre la bibliothèque très minimale que nous proposons, notamment en incluant des éléments permettant de définir des comportements d'objets plus complexes du monde virtuel, comme par exemple des agents autonomes. Cette tâche est un bon test sur le réalisme de notre solution, dont les critères de réussite seront un bon passage à l'échelle et la simplicité des descriptions dans le cadre du *framework*.

Nous aimerions également tester l'adéquation de notre modèle à d'autres familles d'applications distribuées faisant intervenir des éléments hétérogènes et communicants disposant d'autres contraintes matérielles, comme par exemple des applications domotiques ou de robots ludiques [92]. Nous pourrions ainsi étendre de manière pertinente la bibliothèque à de nouveaux protocoles et modèles de communication qui seront peut-être, comme le laisse envisager l'intérêt actuel pour les terminaux de jeux *mobiles*, les supports des distractions de demain.

7.3 Environnement de développement

Nous voudrions dans l'idéal réaliser un environnement de développement complet autour du *framework* que nous avons défini. Celui-ci permettrait d'assembler les composants de l'application, de générer le code spécifique à l'application à partir de cet assemblage, et fournirait des outils de mise au point, comme par exemple rejouer une exécution, à l'aide de scenarii d'événements provenant du réseau ou des interfaces utilisateur.

Dans un futur proche, il sera indispensable de fournir un outil permettant de générer une grande partie du code générique, et capable d'automatiser l'écriture des scripts de configuration, ces tâches étant actuellement fastidieuses et répétitives.

Nous pensons également intégrer un outil de simulation de mauvaises conditions de réseau du style de *NetTool* [60].

Nous voulons ultérieurement étudier les possibilités de réaliser des ana-

lyses statiques s'appuyant sur la sémantique d'exécution du modèle pour identifier et calibrer les caractéristiques de l'application : par exemple, donner une estimation des ressources nécessaires au déploiement de l'application, ce qui permettrait d'éviter le sous-dimensionnement ou le sur-dimensionnement en permettant assez tôt d'avoir une estimation des ressources nécessaires.

Chapitre 8

Annexes

Nous allons présenter dans ce chapitre un complément de l'exemple décrit à la fin du chapitre 6. La première partie de ce chapitre donne la totalité du fichier de configuration utilisé côté serveur pour la réalisation de l'exemple.

La deuxième partie présente les classes Java de description des états ainsi que leurs fabriques, tandis que la troisième présente le code des ordonnançables et leurs fabriques.

Cette présentation a uniquement pour but de compléter la description de notre modélisation et d'illustrer comment le *Fill-In-The-Gaps Toolkit* fonctionne à l'heure actuelle pour le lecteur curieux.

Nous envisageons de générer tout le code présenté au sein de ce chapitre par un outil d'intégration construit autour du *Fill-In-The-Gaps Toolkit*.

8.1 Le fichier de configuration de l'application

STATE game GameStateFactory
KEYDUPLICATEDSTATESETS clients
KEYSUBSTATE lastHostCreated lastMessage newMessage murmureDistance
SUBSTATEINITIALIZE lastHostCreated lastMessage newMessage murmureDistance
BEHAVIORMODELS messageUpdate

```
DUPLICATA clients ClientFactory
KEYDATA x y name
DATAINITIALIZE name.anonymous.string
REPLICATIONMODELS forward hostCreatedReplication

STATE lastHostCreated LastHostConnectedFactory
KEYDATA hostIdentifier
DATAINITIALIZE hostIdentifier.null.string

STATE lastMessage MessageNetworkedStateFactory
KEYDATA message

STATE newMessage MessageStateFactory
KEYDATA message
REPLICATIONMODELS sendNewMessage sendMurmure

STATE murmureDistance MurmureDistanceFactory
KEYDATA distance
DATAINITIALIZE distance.100.int

REPLICATION forward StateChangedReplicationFactory
STATEOBSERVED SELF
SCOPE allButMe
PROTOCOL UDP

REPLICATION hostCreatedReplication HostConnectedReplicationFactory
STATEOBSERVED game.lastHostCreated
STATENEEDED game.lastHostCreated
SCOPE hostCreatedScope
PROTOCOL TCP

BEHAVIOR messageUpdate StateUpdateBehaviorFactory
STATEOBSERVED game.lastMessage
STATENEEDED game.lastMessage game.newMessage
REPLICATION sendNewMessage ChatReplicationFactory
STATEOBSERVED game.newMessage
STATENEEDED game.newMessage
```

SCOPE all
PROTOCOL TCP

REPLICATION sendMurmure MurmureReplicationFactory
STATEOBSERVED game.newMessage
STATENEEDED game.newMessage game.murmureDistance game.clients
SCOPE murmureScope
PROTOCOL TCP

SCOPEDEF allButMe AllButMeScopeFactory

SCOPEDEF all AllScopeFactory

SCOPEDEF hostCreatedScope HostConnectedScopeFactory

SCOPEDEF murmureScope MurmureScopeFactory

PROTOCOLDEF TCP TCPProtocolFactory

PROTOCOLDEF UDP UDPProtocolFactory

8.2 Classes et fabriques d'états

8.2.1 L'état gameState

8.2.1.1 Classe GameState

```
public class GameState extends State {
  private DuplicateStatesSet clients;
  private LastHostConnected lastHostConnected;
  private MessageNetworkedState lastMessage;
  private MessageState newMessage;
  private MurmureDistance murmureDistance;

  public GameState(String path, DataAccessList dataAccessList,
```

```
    String subStatesKeys, StateAccessList substateSetters,
    String datasInitialization, String replicationKeys,
    String behaviorKeys, String substateInitialization,
    String keyDuplicatedStates,
    StateAccessList duplicatedStatesInitializers,
    StateFactory factory) {
  super(path, dataAccessList, subStatesKeys, substateSetters,
      datasInitialization, replicationKeys, behaviorKeys,
      substateInitialization, keyDuplicatedStates,
      duplicatedStatesInitializers, factory);
}
public DuplicateStatesSet getClients() {
  return clients;
}
public void setClients(DuplicateStatesSet clients) {
  this.clients = clients;
}
public synchronized LastHostConnected getLastHostConnected() {
  return lastHostConnected;
}
public synchronized void setLastHostConnected(
    LastHostConnected lastHostConnected) {
  this.lastHostConnected = lastHostConnected;
}
public synchronized MessageNetworkedState getLastMessageState() {
  return lastMessage;
}
public synchronized void setLastMessageState(
    MessageNetworkedState lastMessageState) {
  this.lastMessage = lastMessageState;
}
public synchronized MessageState getNewMessage() {
  return newMessage;
}
public synchronized void setNewMessage(MessageState newMessage) {
  this.newMessage = newMessage;
}
public synchronized MurmureDistance getMurmureDistance() {
  return murmureDistance;
}
```

```
  public synchronized void setMurmureDistance(
      MurmureDistance murmureDistance) {
    this.murmureDistance = murmureDistance;
  }
}
```

8.2.1.2 Classe GameStateFactory

```
public class GameStateFactory extends StateFactory {
  public State createEmptyState() {
    return new GameState(null, null, null, null, null, null, null,
        null, null, null, this);
  }
  public State createState(String key, String keyData,
      String dataInitialize, String replication, String behaviorKeys,
      String keySubstate, String substateInitialization,
      String keyDuplicatedStates) {
    String subStateKeys = subStateDependancies(key, keySubstate);
    StringTokenizer keySubstateTokenizer = new StringTokenizer(
        subStateKeys, " ");
    String lastHostKey = keySubstateTokenizer.nextToken();
    String lastMessageKey = keySubstateTokenizer.nextToken();
    String newMessageKey = keySubstateTokenizer.nextToken();
    String murmureDistanceKey = keySubstateTokenizer.nextToken();
    StateAccessList stateSetterList = new StateAccessList();
    stateSetterList.put(lastHostKey, new StateAccessInState() {
      public StateComponent getState(
          StateCreationObserver stateCreationObserver) {
        return ((GameState) stateCreationObserver)
            .getLastHostConnected();
      }
      public void setState(
          StateCreationObserver stateCreationObserver,
          StateComponent state) {
        ((GameState) stateCreationObserver)
            .setLastHostConnected((LastHostConnected) state);
      }
    });
    stateSetterList.put(lastMessageKey, new StateAccessInState() {
```

```
      public StateComponent getState(
          StateCreationObserver stateCreationObserver) {
        return ((GameState) stateCreationObserver)
            .getLastMessageState();
      }
      public void setState(
          StateCreationObserver stateCreationObserver,
          StateComponent state) {
        ((GameState) stateCreationObserver)
            .setLastMessageState((MessageNetworkedState) state);
      }
    });
    stateSetterList.put(newMessageKey, new StateAccessInState() {
      public StateComponent getState(
          StateCreationObserver stateCreationObserver) {
        return ((GameState) stateCreationObserver).getNewMessage();
      }
      public void setState(
          StateCreationObserver stateCreationObserver,
          StateComponent state) {
        ((GameState) stateCreationObserver)
            .setNewMessage((MessageState) state);
      }
    });
    stateSetterList.put(murmureDistanceKey, new StateAccessInState() {
      public StateComponent getState(
          StateCreationObserver stateCreationObserver) {
        return ((GameState) stateCreationObserver)
            .getMurmureDistance();
      }
      public void setState(
          StateCreationObserver stateCreationObserver,
          StateComponent state) {
        ((GameState) stateCreationObserver)
            .setMurmureDistance((MurmureDistance) state);
      }
    });
    String duplicatedStates = subStateDependancies(key,
        keyDuplicatedStates);
    StringTokenizer st = new StringTokenizer(duplicatedStates, " ");
```

```
    String clientKey = st.nextToken();
    StateAccessList duplicatedStatesInitialization = new StateAccessList();
    duplicatedStatesInitialization.put(clientKey,
        new StateAccessInState() {
          public StateComponent getState(
              StateCreationObserver stateCreationObserver) {
            return ((GameState) stateCreationObserver).getClients();
          }
          public void setState(
              StateCreationObserver stateCreationObserver,
              StateComponent state) {
            ((GameState) stateCreationObserver)
                .setClients((DuplicateStatesSet) state);
          }
        });
    return new GameState(key, null, subStateKeys, stateSetterList,
        null, replication, behaviorKeys, subStateDependancies(key,
            substateInitialization), duplicatedStates,
        duplicatedStatesInitialization, this);
  }
}
```

8.2.2 Les états duplicata clients

8.2.2.1 Classe Client

```
public class Client extends NetworkState {
  public final static int RAY = 10;
  private int x;
  private int y;
  private String name;

  public Client(String path, DataAccessList dataAccessList,
      String subStatesKeys, StateAccessList substateSetters,
      String datasInitialization, String replicationKeys,
      String behaviorKeys, String substateInitialization,
      String keyDuplicatedStates,
      StateAccessList duplicatedStatesInitializers,
      StateFactory factory) {
```

```
    super(path, dataAccessList, subStatesKeys, substateSetters,
        datasInitialization, replicationKeys, behaviorKeys,
        substateInitialization, keyDuplicatedStates,
        duplicatedStatesInitializers, factory);
  }
  public synchronized int getX() {
    return x;
  }
  public synchronized void setX(int x) {
    this.x = x;
  }
  public synchronized int getY() {
    return y;
  }
  public synchronized void setY(int y) {
    this.y = y;
  }
  public synchronized String getName() {
    return name;
  }
  public synchronized void setName(String name) {
    this.name = name;
  }
}
```

8.2.2.2 Classe ClientFactory

```
public class ClientFactory extends StateFactory {
  public State createEmptyState() {
    return new Client(null, null, null, null, null, null, null, null,
        null, null, this);
  }
  public State createState(String key, String keyData,
      String dataInitialize, String replication, String behaviorKeys,
      String keySubstate, String substateInitialization,
      String keyDuplicatedStates) {
    StringTokenizer keyDataTokenizer = new StringTokenizer(keyData,
        " ");
    String xKey = keyDataTokenizer.nextToken();
```

```
    String yKey = keyDataTokenizer.nextToken();
    String nameKey = keyDataTokenizer.nextToken();
    DataAccessList dataSetterList = new DataAccessList();
    dataSetterList.put(xKey, new DataAccess(DataAccess.INTEGER_TYPE) {
      protected void setData(State state, int value) {
        ((Client) state).setX(value);
      }
      protected int getInt(State state) {
        return ((Client) state).getX();
      }
    });
    dataSetterList.put(yKey, new DataAccess(DataAccess.INTEGER_TYPE) {
      protected void setData(State state, int value) {
        ((Client) state).setY(value);
      }
      protected int getInt(State state) {
        return ((Client) state).getY();
      }
    });
    dataSetterList.put(nameKey,
        new DataAccess(DataAccess.STRING_TYPE) {
          protected void setData(State state, String value) {
            ((Client) state).setName(value);
          }
          protected String getString(State state) {
            return ((Client) state).getName();
          }
        });
    return new Client(key, dataSetterList, null, null,
        dataInitialize, replication, behaviorKeys,
        substateInitialization, null, null, this);
  }
}
```

8.2.3 L'état lastMessage

8.2.3.1 Classe MessageNetworkedState

```
public class MessageNetworkedState extends NetworkState {
```

```
  private String message;

  public MessageNetworkedState(String path,
      DataAccessList dataAccessList, String subStatesKeys,
      StateAccessList substateSetters, String datasInitialization,
      String replicationKeys, String behaviorKeys,
      String substateInitialization, String keyDuplicatedStates,
      StateAccessList duplicatedStatesInitializers,
      StateFactory factory) {
    super(path, dataAccessList, subStatesKeys, substateSetters,
        datasInitialization, replicationKeys, behaviorKeys,
        substateInitialization, keyDuplicatedStates,
        duplicatedStatesInitializers, factory);
  }
  public synchronized String getMessage() {
    return message;
  }
  public synchronized void setMessage(String message) {
    this.message = message;
  }
}
```

8.2.3.2 Classe MessageNetworkedStateFactory

```
public class MessageNetworkedStateFactory extends StateFactory {
  public State createEmptyState() {
    return new MessageNetworkedState(null, null, null, null, null,
        null, null, null, null, null, this);
  }
  public State createState(String key, String keyData,
      String dataInitialize, String replication, String behaviorKeys,
      String keySubstate, String substateInitialization,
      String keyDuplicatedStates) {
    StringTokenizer keyDataTokenizer = new StringTokenizer(keyData,
        " ");
    String messageKey = keyDataTokenizer.nextToken();
    DataAccessList dataSetterList = new DataAccessList();
    dataSetterList.put(messageKey, new DataAccess(
        DataAccess.STRING_TYPE) {
```

```
      protected void setData(State state, String value) {
        ((MessageNetworkedState) state).setMessage(value);
      }
      protected String getString(State state) {
        return ((MessageNetworkedState) state).getMessage();
      }
    });
    return new MessageNetworkedState(key, dataSetterList, null, null,
        dataInitialize, replication, behaviorKeys, null, null, null,
        this);
  }
}
```

8.2.4 L'état newMessage

8.2.4.1 Classe MessageState

```
public class MessageState extends State {
  private String message;

  public MessageState(String path, DataAccessList dataAccessList,
      String subStatesKeys, StateAccessList substateSetters,
      String datasInitialization, String replicationKeys,
      String behaviorKeys, String substateInitialization,
      String keyDuplicatedStates,
      StateAccessList duplicatedStatesInitializers,
      StateFactory factory) {
    super(path, dataAccessList, subStatesKeys, substateSetters,
        datasInitialization, replicationKeys, behaviorKeys,
        substateInitialization, keyDuplicatedStates,
        duplicatedStatesInitializers, factory);
  }
  public synchronized String getMessage() {
    return message;
  }
  public synchronized void setMessage(String message) {
    this.message = message;
  }
}
```

8.2.4.2 Classe MessageStateFactory

```
public class MessageStateFactory extends StateFactory {
  public State createEmptyState() {
    return new MessageState(null, null, null, null, null, null, null,
        null, null, null, this);
  }
  public State createState(String key, String keyData,
      String dataInitialize, String replication, String behaviorKeys,
      String keySubstate, String substateInitialization,
      String keyDuplicatedStates) {
    StringTokenizer keyDataTokenizer = new StringTokenizer(keyData,
        " ");
    String messageKey = keyDataTokenizer.nextToken();
    DataAccessList dataSetterList = new DataAccessList();
    dataSetterList.put(messageKey, new DataAccess(
        DataAccess.STRING_TYPE) {
      protected void setData(State state, String value) {
        ((MessageState) state).setMessage(value);
      }
      protected String getString(State state) {
        return ((MessageState) state).getMessage();
      }
    });
    return new MessageState(key, dataSetterList, null, null,
        dataInitialize, replication, behaviorKeys, null, null, null,
        this);
  }
}
```

8.2.5 L'état lastHostConnected

8.2.5.1 Classe LastHostConnected

```
public class LastHostConnected extends State {
  private String hostIdentifier;

  public LastHostConnected(String path,
      DataAccessList dataAccessList, String subStatesKeys,
```

```
      StateAccessList substateSetters, String datasInitialization,
      String replicationKeys, String behaviorKeys,
      String substateInitialization, String keyDuplicatedStates,
      StateAccessList duplicatedStatesInitializers,
      StateFactory factory) {
    super(path, dataAccessList, subStatesKeys, substateSetters,
        datasInitialization, replicationKeys, behaviorKeys,
        substateInitialization, keyDuplicatedStates,
        duplicatedStatesInitializers, factory);
  }
  public synchronized String getHostIdentifier() {
    return hostIdentifier;
  }
  public synchronized void setHostIdentifier(String hostIdentifier) {
    this.hostIdentifier = hostIdentifier;
  }
}
```

8.2.5.2 Classe LastHostConnectedFactory

```
public class LastHostConnectedFactory extends StateFactory {
  public State createEmptyState() {
    return new LastHostConnected(null, null, null, null, null, null,
        null, null, null, null, this);
  }
  public State createState(String key, String keyData,
      String dataInitialize, String replication, String behaviorKeys,
      String keySubstate, String substateInitialization,
      String keyDuplicatedStates) {
    StringTokenizer keyDataTokenizer = new StringTokenizer(keyData,
        " ");
    // définition des initialisateurs d'attributs de types simples(non états)
    String hostIdentifierKey = keyDataTokenizer.nextToken();
    DataAccessList dataSetterList = new DataAccessList();
    dataSetterList.put(hostIdentifierKey, new DataAccess(
        DataAccess.STRING_TYPE) {
      protected void setData(State state, String value) {
        ((LastHostConnected) state).setHostIdentifier(value);
      }
```

```
      protected String getString(State state) {
        return ((LastHostConnected) state).getHostIdentifier();
      }
    });
    return new LastHostConnected(key, dataSetterList, null, null,
        dataInitialize, null, null, null, null, null, this);
  }
}
```

8.2.6 L'état murmureDistance

8.2.6.1 Classe MurmureDistance

```
public class MurmureDistance extends State {
  private int distance;

  public MurmureDistance(String path, DataAccessList dataAccessList,
      String subStatesKeys, StateAccessList substateSetters,
      String datasInitialization, String replicationKeys,
      String behaviorKeys, String substateInitialization,
      String keyDuplicatedStates,
      StateAccessList duplicatedStatesInitializers,
      StateFactory factory) {
    super(path, dataAccessList, subStatesKeys, substateSetters,
        datasInitialization, replicationKeys, behaviorKeys,
        substateInitialization, keyDuplicatedStates,
        duplicatedStatesInitializers, factory);
    // TODO Auto-generated constructor stub
  }
  public synchronized int getDistance() {
    return distance;
  }
  public synchronized void setDistance(int distance) {
    this.distance = distance;
  }
}
```

8.2.6.2 Classe MurmureDistanceFactory

```
public class MurmureDistanceFactory extends StateFactory {
  public State createEmptyState() {
    return new MurmureDistance(null, null, null, null, null, null,
        null, null, null, null, this);
  }
  public State createState(String key, String keyData,
      String dataInitialize, String replication, String behaviorKeys,
      String keySubstate, String substateInitialization,
      String keyDuplicatedStates) {
    StringTokenizer keyDataTokenizer = new StringTokenizer(keyData,
        " ");
    String distanceKey = keyDataTokenizer.nextToken();
    DataAccessList dataSetterList = new DataAccessList();
    dataSetterList.put(distanceKey, new DataAccess(
        DataAccess.INTEGER_TYPE) {
      protected void setData(State state, int value) {
        ((MurmureDistance) state).setDistance(value);
      }
      protected int getInt(State state) {
        return ((MurmureDistance) state).getDistance();
      }
    });
    return new MurmureDistance(key, dataSetterList, null, null,
        dataInitialize, replication, behaviorKeys, null, null, null,
        this);
  }
}
```

8.3 Classes et fabriques d'ordonnançables

8.3.1 Le modèle de réplication forward

8.3.1.1 Classe StateChangedReplication

```
public class StateChangedReplication extends Replication {
```

```
  public StateChangedReplication(State state,
      StateAccessList stateSetters, String scope, String protocol,
      String statesObserved) {
    super(state, stateSetters, scope, protocol, statesObserved);
  }
  protected boolean realized() {
    return true;
  }
}
```

8.3.1.2 Classe StateChangedReplicationFactory

```
public class StateChangedReplicationFactory implements
    ReplicationFactory {
  public Replication create(State state, String scope,
      String protocol, String statesNeeded, String statesObserved) {
    return new StateChangedReplication(state, null, scope, protocol,
        statesObserved);
  }
}
```

8.3.2 Le modèle de réplication hostCreatedReplication

8.3.2.1 Classe HostConnectedReplication

```
public class HostConnectedReplication extends Replication {
  private LastHostConnected lastHostConnected;

  public HostConnectedReplication(State state,
      StateAccessList stateSetters, String scope, String protocol,
      String statesObserved) {
    super(state, stateSetters, scope, protocol, statesObserved);
  }
  protected boolean realized() {
    return true;
  }
  public synchronized LastHostConnected getLastHostConnected() {
    return lastHostConnected;
```

```
  }
  public synchronized void setLastHostConnected(
      LastHostConnected lastHostConnected) {
    this.lastHostConnected = lastHostConnected;
  }
}
```

8.3.2.2 Classe HostConnectedReplicationFactory

```
public class HostConnectedReplicationFactory implements
    ReplicationFactory {
  public Replication create(State state, String scope,
      String protocol, String statesNeeded, String statesObserved) {
    StringTokenizer statesNeededTokenizer = new StringTokenizer(
        statesNeeded, " ");
    String hostKey = statesNeededTokenizer.nextToken();
    StateAccessList stateSetterList = new StateAccessList();
    stateSetterList.put(hostKey, new SetStateInReplication() {
      public void setState(
          StateCreationObserver stateCreationObserver,
          StateComponent state) {
        ((HostConnectedReplication) stateCreationObserver)
            .setLastHostConnected((LastHostConnected) state);
      }
    });
    return new HostConnectedReplication(state, stateSetterList,
        scope, protocol, statesObserved);
  }
}
```

8.3.3 L'ordonnançable messageUpdate

8.3.3.1 Classe StateUpdateBehavior

```
public class StateUpdateBehavior extends Behavior {
  private MessageState newMessage;
  private MessageNetworkedState lastMessage;
```

```
  public StateUpdateBehavior(State owner,
      StateAccessList stateSetterList, String statesObserved) {
    super(owner, stateSetterList, statesObserved);
    super.callback = new Callback(this) {
      protected void execute() {
        newMessage.setMessage(lastMessage.getMessage());
        update(newMessage);
      }
    };
  }
  protected boolean realized() {
    return true;
  }
  public synchronized void setLastMessage(
      MessageNetworkedState lastMessage) {
    this.lastMessage = lastMessage;
  }
  public synchronized void setNewMessage(MessageState newMessage) {
    this.newMessage = newMessage;
  }
}
```

8.3.3.2 Classe StateUpdateBehaviorFactory

```
public class StateUpdateBehaviorFactory implements BehaviorFactory
{
  public Behavior create(State state, String statesNeeded,
      String statesObserved) {
    StringTokenizer statesNeededTokenizer = new StringTokenizer(
        statesNeeded, " ");
    String lastMessageKey = statesNeededTokenizer.nextToken();
    String newMessageKey = statesNeededTokenizer.nextToken();
    StateAccessList stateSetterList = new StateAccessList();
    stateSetterList.put(newMessageKey, new SetStateInReplication() {
      public void setState(
          StateCreationObserver stateCreationObserver,
          StateComponent state) {
        ((StateUpdateBehavior) stateCreationObserver)
            .setNewMessage((MessageState) state);
```

```
      }
    });
    stateSetterList.put(lastMessageKey, new SetStateInReplication() {
      public void setState(
          StateCreationObserver stateCreationObserver,
          StateComponent state) {
        ((StateUpdateBehavior) stateCreationObserver)
            .setLastMessage((MessageNetworkedState) state);
      }
    });
    return new StateUpdateBehavior(state, stateSetterList,
        statesObserved);
  }
}
```

8.3.4 Le modèle de réplication sendNewMessage

8.3.4.1 Classe ChatReplication

```
public class ChatReplication extends Replication {
  private MessageState newMessage;

  public ChatReplication(State state, StateAccessList stateSetters,
      String scope, String protocol, String statesObserved) {
    super(state, stateSetters, scope, protocol, statesObserved);
  }
  public synchronized void setNewMessage(MessageState newMessage) {
    this.newMessage = newMessage;
  }
  protected boolean realized() {
    return ((newMessage.getMessage() != null) && !newMessage
        .getMessage().startsWith(Serveur.MURMURE_CMD));
  }
}
```

8.3.4.2 Classe ChatReplicationFactory

```
public class ChatReplicationFactory implements ReplicationFactory
```

```
{
  public Replication create(State state, String scope,
      String protocol, String statesNeeded, String statesObserved) {
    StringTokenizer statesNeededTokenizer = new StringTokenizer(
        statesNeeded, " ");
    String newMessageKey = statesNeededTokenizer.nextToken();
    StateAccessList stateSetterList = new StateAccessList();
    stateSetterList.put(newMessageKey, new SetStateInReplication() {
      public void setState(
          StateCreationObserver stateCreationObserver,
          StateComponent state) {
        ((ChatReplication) stateCreationObserver)
            .setNewMessage((MessageState) state);
      }
    });
    return new ChatReplication(state, stateSetterList, scope,
        protocol, statesObserved);
  }
}
```

8.3.5 Le modèle de réplication sendMurmure

8.3.5.1 Classe MurmureReplication

```
public class MurmureReplication extends Replication {
  private MessageState newMessage;
  private MurmureDistance murmureDistance;
  private DuplicateStatesSet clients;

  public MurmureReplication(State state,
      StateAccessList stateSetters, String scope, String protocol,
      String statesObserved) {
    super(state, stateSetters, scope, protocol, statesObserved);
  }
  public synchronized void setNewMessage(MessageState newMessage) {
    this.newMessage = newMessage;
  }
  public synchronized MurmureDistance getMurmureDistance() {
    return murmureDistance;
```

```
  }
  public synchronized void setMurmureDistance(
      MurmureDistance murmureDistance) {
    this.murmureDistance = murmureDistance;
  }
  public synchronized MessageState getNewMessage() {
    return newMessage;
  }
  public synchronized DuplicateStatesSet getClients() {
    return clients;
  }
  public synchronized void setClients(DuplicateStatesSet clients) {
    this.clients = clients;
  }
  protected boolean realized() {
    return ((newMessage.getMessage() != null) && newMessage
        .getMessage().startsWith(Serveur.MURMURE_CMD));
  }
}
```

8.3.5.2 Classe MurmureReplicationFactory

```
public class MurmureReplicationFactory implements
ReplicationFactory {
  public Replication create(State state, String scope,
      String protocol, String statesNeeded, String statesObserved) {
    StringTokenizer statesNeededTokenizer = new StringTokenizer(
        statesNeeded, " ");
    String newMessageKey = statesNeededTokenizer.nextToken();
    String murmureDistanceKey = statesNeededTokenizer.nextToken();
    String clientsKey = statesNeededTokenizer.nextToken();
    StateAccessList stateSetterList = new StateAccessList();
    stateSetterList.put(newMessageKey, new SetStateInReplication() {
      public void setState(
          StateCreationObserver stateCreationObserver,
          StateComponent state) {
        ((MurmureReplication) stateCreationObserver)
            .setNewMessage((MessageState) state);
      }
```

```
    });
    stateSetterList.put(murmureDistanceKey,
        new SetStateInReplication() {
          public void setState(
              StateCreationObserver stateCreationObserver,
              StateComponent state) {
            ((MurmureReplication) stateCreationObserver)
                .setMurmureDistance((MurmureDistance) state);
          }
        });
    stateSetterList.put(clientsKey, new SetStateInReplication() {
      public void setState(
          StateCreationObserver stateCreationObserver,
          StateComponent state) {
        ((MurmureReplication) stateCreationObserver)
            .setClients((DuplicateStatesSet) state);
      }
    });
    return new MurmureReplication(state, stateSetterList, scope,
        protocol, statesObserved);
  }
}
```

8.3.5.3 Classe MurmureScope

```
public class MurmureScope extends Scope {
  public MurmureScope(String key) {
    super(key);
  }
  public void send(Replication replication, State state,
      CommunicationProperties protocol) {
    MurmureReplication murmureReplication = (MurmureReplication) replication;
    DuplicateStatesSet clientStates = murmureReplication.getClients();
    MurmureDistance murmureDistance = murmureReplication
        .getMurmureDistance();
    String message = ((MessageState) state).getMessage();
    StringTokenizer st = new StringTokenizer(message, " ");
    st.nextToken();
    String senderKey = st.nextToken();
```

```
    Client sender = null;
    for (int i = 0; i < clientStates.getSize(); i++) {
      Client client = (Client) clientStates.getDuplicata(i);
      if (client.getIdentifier().equals(senderKey)) {
        sender = client;
        break;
      }
    }
    for (int i = 0; i < clientStates.getSize(); i++) {
      Client client = (Client) clientStates.getDuplicata(i);
      if ((client.getX() >= (sender.getX() - murmureDistance
          .getDistance()))
          && (client.getX() <= (sender.getX() + murmureDistance
              .getDistance()))
          && (client.getY() >= (sender.getY() - murmureDistance
              .getDistance()))
          && (client.getY() <= (sender.getY() + murmureDistance
              .getDistance()))) {
        Recipient recipient = hosts.findRecipient(client
            .getIdentifier());
        recipient.send(state, protocol);
      }
    }
  }
}
```

Bibliographie

[1] Aarhus, L., Holmqvist, K., et Kirkengen, M. Generalized Two-Tier Relevance Filtering of Computer Game Update events. In *Proceedings of ACM NetGames* (Avril 2002).

[2] Aggarwal, S., Banavar, H., Khandelwal, A., Mukherjee, S., et Rangarajan, S. Accuracy in Dead-Reckoning Based Distributed Multi-Player Games. In *Proceedings of ACM NetGames* (Août 2004).

[3] Armitage, G. Lag Over 150 Millisecond is Unacceptable, Mai 2001. disponible sur : `http ://gja.space4me.com/things/quake3-latency-051701.html`.

[4] Arnold, K., Scheifler, R., Waldo, J., O'Sullivan, B., Wollrath, A., O'Sullivan, B., et Wollrath, A. *Jini Specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[5] Bartle, R. Hearts, Clubs, Diamond, Spades : Players Who Suit Muds. disponible sur : `http ://www.mud.co.uk/richard/hcds.htm`.

[6] Baughman, N. E., et Levine, B. N. Cheat-Proof Playout for Centralized and Distributed Online Games. In *Proceedings INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies* (2001).

[7] Beck, K. *Extreme Programming Explained : Embrace Change*, première éd. Addison-Wesley Professional, 1999.

[8] Beck, K. *Crystal Clear : A Human-Powered Methodology for Small Teams*, première éd. Addison-Wesley Professional, 2004.

[9] Beigbeder, T., Coughlan, R., Lusher, C., Plunkett, J., Agu, E., et Claypool, M. The Effects of Loss and Latency on User Performance in Unreal Tournament 2003. In *Proceedings of ACM NetGames* (Août 2004).

[10] Bernier, Y. W. Latency Compensating Methods in Client/Server In-Game Protocol Design and Optimization. In *Proceedings of the Game Developers Conferences* (Mars 2000).

[11] Berry, G. The Esterel v5 Language Primer. disponible sur : `ftp ://ftp-sop.inria.fr/meije/esterel/papers/primer.pdf`.

[12] Bharambe, A. R., Rao, S., et Seshan, S. Mercury : A Scalable Publish-Subscribe System for Internet Games. In *Proceedings of ACM NetGames* (Avril 2002).

[13] Boehm, B. A Spiral Model of Software Development and Enhancement. *SIGSOFT Softw. Eng. Notes 11*, 4 (1986).

[14] Bosser, A.-G. Sémantique opérationnelle du langage Scol, 2002. Rapport d'avancement du projet RNTL EDICA. disponible sur : `http ://www.pps.jussieu.fr/~bosser/papers/`.

[15] Bosser, A.-G. Massively Multi-player Games, Matching Game Design with Technical Design. In *Proceedings of ACM Advances in Computer Entertainment* (Juin 2004).

[16] Bosser, A.-G. Massively Multi-player Games, Matching Game Design with Technical Design. Version augmentée. Prix International Imagina 2005 du Jeune Chercheur dans le domaine des jeux-vidéos, disponible sur : `http ://www.pps.jussieu.fr/~bosser/papers/`.

[17] Bosser, A.-G. Replication Model for Designing Multi-player Games Interactions. A Paraître, *Proceedings of CGAMES 2005* (Novembre 2005).

[18] Bosser, A.-G., et Alberti, F. L'expérience scol, un langage pour des applications Internet multi-utilisateurs. In *Journées Francophones des Langages Applicatifs* (Février 2003), INRIA.

[19] Bouraqadi-Saâdani, N. M. N., et Ledoux, T. Le point sur la programmation par aspects. In *Journal Technique et science informatiques* (2001), vol. 20-4, Hermès.

[20] Boussinot, F. *Java Fair Threads*. Tech. Rep. RR-4139, INRIA, 2001. `http ://www.inria.fr/rrrt/rr-4139.html`.

[21] Boussinot, F., Susini, J.-F., Tran, F. D., et Hazard, L. A Reactive Behavior Framework for Dynamic Virtual Worlds. In *Web3D '01 : Proceedings of the sixth international conference on 3D Web technology* (2001).

[22] Castronova, E. On Virtual Economies. In *CESifo Working Paper Series No. 752* (Juillet 2002).

[23] Chen, Y.-C., Chen, P. S., Song, R., et Korba, L. Online Gaming Crime and Security Issue - Cases and Countermeasures from Taiwan. In *Proceedings of the 2nd Annual Conference on Privacy, Security and Trust (PST'2004)* (Octobre 2004).

[24] City of Heroes. `http ://www.cityofheroes.com/`.

[25] Cizault, G. *IPv6, Théorie et Pratique*, troisième éd. O'Reilly, 2002.

[26] Coulouris, G., Dollimore, J., et Kindberg, T. *Distributed Systems : Concepts and Design*, troisième éd. Addison-Wesley, 2001.

[27] Cronin, E., Filstrup, B., Kurc, A. R., et Jamin, S. An Efficient Synchronization Mechanism for Mirrored Game Architectures. In *Proceedings of ACM NetGames* (Avril 2002).

[28] Deloura, M., et Treglia, D. *Game Programming Gems 3*. Delmar Thomson Learning, 2002. Ouvrage Collectif.

[29] Demanchy, T. Extreme Game Development : Right on Time, Every Time, Juillet 2003. available on `http ://www.gamasutra.com`.

[30] Diot, C., et Gautier, L. Design and Evaluation of Mimaze, a Multi-Player Game on the Internet. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems* (1998).

[31] Diot, C., et Gautier, L. A Distributed Architecture for Multiplayer Interactive Applications on the Internet. In *IEEE Networks magazine, vol. 17, n.4* (1999).

[32] DOD-STD-2167A. *Defense Systems Software Development*, 1988. Standard de développement.

[33] Esterel technologies. `http ://www.esterel-technologies.com`.

[34] Everquest. `http ://eqlive.station.sony.com/`.

[35] Extreme Programming. `http ://www.extremeprogramming.org/`.

[36] Fabre, Y. A Framework for Mobile-Agents Embodied in X3D Networked Virtual Environment. In *Web3D '03 : Proceeding of the eighth international conference on 3D Web technology* (2003), ACM Press.

[37] Faisstnauer, C., Schmalstieg, D., et Purgathofer, W. Scheduling for Very Large Virtual Environments and Networked Games using Visibilities and Priorities. In *Proceedings of the DIS-RT conference* (2000).

[38] Fayad, M. E., Schmidt, D. C., et Johnson, R. E. *Building Application Frameworks*. Wiley, 1999.

[39] Ferguson, M., et Ballbach, M. Product review : Massively multiplayer online game middleware, Janvier 2003. disponible sur : `http ://www.gamasutra.com`.

[40] Fiedler, S., Wallner, M., et Weber, M. A Communication Architecture for Massive Multiplayer Games. In *Proceedings of ACM NetGames* (Avril 2002).

[41] Firor, M. Postmortem : Mythic's dark ages of camelot. Game Developer magazine, Février 2002. disponible sur : `http ://www.gamasutra.com/features/`.

[42] Foo, C. Y., et Koivisto, E. M. Defining Grief-Play in MMORPGs : Player and Developer Perceptions. In *Proceedings of ACM Advances in Computer Entertainment* (Juin 2004).

[43] Fowler, M., Beck, K., Brant, J., Opdyke, W., et Roberts, D. *Refactoring : Improving the Design of Existing Code*, première éd. Addison-Wesley Professional, 1999.

[44] Frecon, E., et Stenius, M. Dive : A Scaleable Network Architecture for Distributed Virtual Environments. In *Distributed Systems Engineering Journal* (1998), vol. 5, pp. 91–100.

[45] Funkhouser, T. Network Topologies for Scalable Multi-User Virtual Environments. In *Proceedings of the 1996 Virtual Reality Annual International Symposium (VRAIS 96)* (1996), IEE Computer Society.

[46] Gamma, E., Helm, R., Johnson, R., et Vlissides, J. *Design Patterns : Elements of Reusable Object-Oriented Software*, première éd. Addison-Wesley Professional, 1995.

[47] Godager, G. *Developing Online Games, an Insider's guide*. New Riders, Mars 2003, ch. Anarchy Online Post-Mortem, pp. 291–314. Livre de Jessica Mulligan et Bridgette Patrowsky.

[48] Griwodz, C. State Replication for Multiplayer Games. In *Proceedings of ACM NetGames* (Avril 2002).

[49] Guo, K., Mukherjee, S., Rangarajan, S., et Paul, S. A Fair Message Exchange Framework for Distributed Multi-Player Games. In *Proceedings of ACM NetGames* (Mai 2003).

[50] Hazard, L., Susini, J.-F., et Boussinot, F. The Junior Reactive Kernel. Manuel de référence, Centre de Mathématiques Appliquées (INRIA et École des Mines de Paris), 1999.

[51] Half-life. `http ://www.planethalflife.com`.

[52] Hu, S.-Y., et Liao, G.-M. Scalable Peer-to-Peer Networked Virtual Environment. In *Proceedings of ACM NetGames* (Août 2004).

[53] Horizons, Empire of Istaria. http ://www.istaria.com/.

[54] Igda(International Game Developers Association) Online Games White paper, Mars 2003.

[55] Iimura, T., Hazeyama, H., et Kadobayashi, Y. Zoned Federation of Game Servers : a Peer-to-Peer Approach to Scalable Multi-Player Online Games. In *Proceedings of ACM NetGames* (Août 2004).

[56] ISO/IEC 12207. *Information Technology, Software life-cycle process*, 1995. Standard International.

[57] Jacobson, I., Booch, G., et Rumbaugh, J. *The Unified Software Development Process*, première éd. Addison-Wesley Professional, 1999.

[58] Jakobsson, M., et Taylor, T. The Sopranos Meets Everquest, Social Networking in Massively Multiplayer Online Games. In *Melbourne DAC - the 5th International Digital Arts and Culture Conference* (Septembre 2004).

[59] Jouni Smed, T. K., et Hakonen, H. Aspects of Networking in Multiplayer Computer Games. In *The Electronic Library* (2002), vol. 20, pp. 87–97.

[60] Kirmse, A. *Game Programming Gem's 3*, Charles River Media. 2002, ch. A Network Monitoring and Simulation Tool, pp. 557–560. Édition rassemblée par Dante Treglia.

[61] Kolo, C., et Baur, T. Living a Virtual Life : Social Dynamics of Online Gaming. In *Game Studies* (Novembre 2004).

[62] Lao, L., Cui, J.-H., Gerla, M., et Maggiorini, D. A Comparative Study of Multicast Protocols : Top Bottom or in the Middle ? In *8th IEEE Global Internet Symposium* (Mars 2005).

[63] Leroy, X., et Weis, P. *Le Langage Caml*, deuxième éd. Dunod, 1999.

[64] Levine, B. N., Crowcroft, J., Diot, C., Garcia-Luna-Aceves, J. J., et Kurose, J. F. Consideration of Receiver Interest for IP Multicast Delivery. In *INFOCOM (2)* (2000), pp. 470–479.

[65] Léty, E., Turletti, T., et Baccelli, F. Cell-Based Multicast Grouping in Large-Scale Virtual Environments (poster session , extended abstract). *SIGMETRICS Perform. Eval. Rev. 28*, 1 (2000).

[66] Maes, P. Concepts and Experiments in Computational Reflection. In *OOPSLA '87 : Conference proceedings on Object-oriented programming systems, languages and applications* (1987), ACM Press.

[67] Martin, J. *Rapid Application Development*, première éd. Macmillan Coll Div, 1991.

[68] Martin Mauve, S. F., et Widmer, J. A Generic Proxy System for Networked Computer Games. In *Proceedings of ACM NetGames* (Avril 2002).

[69] Milner, R. *Communication and Concurrency*. Prentice-Hall, Inc., 1989.

[70] Morse, K. L. Interest Management in Large-Scale Distributed Simulations. Tech. Rep. ICS-TR-96-27, Department of Information and Computer Science, University of California, Irvine, 1996.

[71] Moser, L. E., Amir, Y., Melliar-Smith, P. M., et Agarwal, D. A. Extended Virtual Synchrony. In *The 14th IEEE International Conference on Distributed Computing Systems (ICDCS)* (1994).

[72] Mulligan, J., et Patrovsky, B. *Developing Online Games : an Insider's Guide*. New Riders, 2002.

[73] Nanale, E., et Wyman, M. Prototyping for fun and profit. In *Proceedings of the Game Developers Conferences* (Mars 2000).

[74] Natkin, S. Architectures et technologies informatiques pour jouer à un million de joueurs. *Les Cahiers du Numérique 4(2)*. Hermes, 2003.

[75] Natkin, S. *Jeux Vidéo et médias du XXI^e siècle*. Vuibert, 2004.

[76] Neocron. `http ://www.neocron.com`.

[77] Object Management Group. *UML specification, version 2.0*, 2005.

[78] Okanda, P., et Blair, G. OpenPING : A Reflective Middleware for the Construction of Adaptive Networked Game Applications. In *Proceedings of ACM NetGames* (Août 2004).

[79] Oliveira, M. Virtual Environment System Layered Object Model. In *Proceedings of ACM Advances in Computer Entertainment* (Juin 2004).

[80] Pantel, L., et Wolf, L. C. On the impact of delay on real-time multiplayer games. In *Proceedings of the Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)* (Mai 2002).

[81] Pantel, L., et Wolf, L. C. On the Suitability of Dead-Reckoning Schemes for Games. In *Proceedings of ACM NetGames* (Avril 2002).

[82] Pellegrino, J. D., et Dovrolis, C. Bandwidth Requirement and State Consistency in Three Multiplayer Game Architectures. In *Proceedings of ACM NetGames* (Mai 2003).

[83] Platform for Interactive Networked Games (PING) IST - 1999 - 11488j. Description du projet disponible sur : `http ://www.cordis.lu/ist/`, page principale du projet : `http ://www.pingproject.org/`.

[84] Quake. `http ://www.planetquake.com`.

[85] Quax, P., Monsieurs, P., Lamotte, W., Vleeschauwer, D. D., et Degrande, N. Objective and Subjective Evaluation of the Influence of Small Amounts of Delay and Jitter on a Recent First Person Shooter Game. In *Proceedings of ACM NetGames* (Août 2004).

[86] Renderware. `http ://www.renderware.com/`.

[87] Richard, N. *Description de comportements d'agents autonomes évoluant dans des mondes virtuels.* Thèse d'université, École Nationale Supérieure des Télécommunications, 1996.

[88] Java Remote Method Invocation (Java RMI). documentation : `http ://java.sun.com/products/jdk/rmi/`.

[89] Royce, W. W. Managing the Development of Large Software Systems : Concepts and Techniques. In *Technical Papers of Western Electronic Show and Convention (WesCon)* (Août 1970).

[90] Schmidt, D., Stal, M., Rohnert, H., et Buschmann, F. *Pattern-Oriented Software Architecture : Patterns for Concurrent and Networked Objects.* Wiley, 2000.

[91] Sheldon, N., Girard, E., Borg, S., Claypool, M., et Agu, E. The Effect of Latency on User Performance in Warcraft III. In *Proceedings of ACM NetGames* (Mai 2003).

[92] Shinozaki, K., Sakamoto, H., Tanaka, T., Nakatsu, R. Communication and Control of a Home Robot Using a Mobile Phone. In

PCM 2005 : 6th Pacific Rim Conference on Multimedia (2005), Springer Verlag.

[93] SINGHAL, S., ET ZYDA, M. *Networked Virtual Environments : Design and Implementation.* ACM Press/Addison-Wesley Publishing Co., 1999.

[94] SINGHAL, S. K. *Effective Remote Modeling in Large-Scale Distributed Simulation and Visualization Environments.* PhD thesis, Standford University, 2002.

[95] Starwars galaxies. `http ://starwarsgalaxies.station.sony.com`.

[96] THOR, A. *Massively Multiplayer Game Development.* Charles River Media, 2003. Ouvrage Collectif.

[97] TRANG, F. D., ET GERODOLLE, A. An Object-Oriented Framework for Large-Scale Networked Virtual Environments. In *Proceedings of the 6th International Euro-Par Conference, Munich, Germany* (Septembre 2000), Springer Verlag.

[98] VERLAGUET, J., ET CHAILLOUX, E. Fair Threads Migrations for Objective Caml. In *Proceedings of the 3rd International Workshop on High-level Parallel Programming and Applications* (2005).

[99] VIRTOOLS. Prototyping in DEV. Whitepaper.

[100] VRML Consortium Incorporated. The Virtual Reality Modeling Language, 1997.

[101] WADLER, P. Why No One Uses Functional Languages. In *SIGPLAN Notices, 33(8) :23–27* (1998).

[102] Wanadoo, rapport d'activité 2001 aux actionnaires, Février 2002.

[103] YAN, J. J., ET CHOI, H.-J. Security Issues in Online Games. In *International Conference on Application and Development of Computer Games in the 21st Century* (Novembre 2001).

Glossaire

Aimbot: dans un FPS, programme corrigeant les commandes de l'utilisateur lui permettant ainsi de toujours viser juste.

Avatar: incarnation d'un joueur dans le monde virtuel dans lequel se déroule le jeu.

Dead-reckoning: technique qui consiste à extrapoler la nouvelle valeur d'un état du jeu, dont la mise à jour tarde à arriver ou s'est perdue dans le réseau, à partir de ses valeurs précédentes. Bien sûr, la valeur extrapolée peut se révéler finalement incorrecte auquel cas il faudra la corriger de la manière la plus discrète possible. Il faut donc mettre au point deux algorithmes complémentaires, l'un pour la prédiction, l'autre pour la correction éventuelle de cette prédiction..

Déterminisme: un modèle est dit déterministe si il est possible de prédire à l'avance les résultats obtenus en sortie pour chaque entrée.

Framework: à l'inverse d'une bibliothèque, qui est un ensemble de briques logicielles utilisables par le développeur qui écrit une application, un *framework* décrit l'application, laissant au développeur la tâche de définir les briques logicielles spécifiques à l'application.

Game-design: jargon de l'industrie du jeu-vidéo. Désigne la spécification fonctionnelle du *game-play* d'un jeu.

Game-designer: jargon de l'industrie du jeu-vidéo. Personne dont le métier est de réaliser des *game-designs*.

Game-play: jargon de l'industrie du jeu-vidéo. Dans le cadre des jeux massivement multi-joueurs, le *game-play* est à la fois la description des règles du jeu, mais aussi la spécification de la manière dont chaque joueur interagit avec le monde virtuel dans lequel ce jeu se déroule.

Grief-play: proche du concept de *player-killer*, ce terme désigne le fait d'adopter un style de jeu propre à gâcher intentionnellement l'expérience des autres joueurs.

Hardcore-gamer: dans le jargon du jeu vidéo, désigne un joueur qui dédie la plupart de son temps libre à son loisir préféré.

Jeu Massivement Multi-Joueurs: jeu-vidéo où des centaines voire des milliers de participants évoluent simultanément dans un monde virtuel persistant sur Internet. Plus que le nombre de participants simultanés, qui varie selon les définitions, c'est le caractère persistant de ces jeux qui les différencient des autres jeux en ligne : un jeu massivement multi-joueurs est un monde virtuel, qui est accessible en permanence, et qui continue à évoluer même quand le joueur n'y est pas connecté. Ce dernier y incarne un (ou plusieurs) avatar qui est le même d'une connexion à l'autre, et qui évolue au gré de ses peregrinations dans l'univers du jeu.

Kill steal: dans le jargon du jeu-vidéo, technique consistant à attendre qu'un joueur aie presque fini d'achever un PNJ pour donner le coup fatal afin d'emporter l'expérience ou les récompenses associées.

Killer-application: jargon de marketing technologique. Désigne une application réalisée à l'aide d'une technologie donnée permettant de rendre ladite technologie indispensable au consommateur.

Loot steal: dans le jargon du jeu-vidéo, technique consistant dans à ramasser très vite les récompenses associées à un succès d'un joueur lors d'un combat, avant que ce dernier aie eu le temps de le faire.

Personnage Non-Joueur: jargon de l'industrie du jeu-vidéo, souvent désigné par l'acronyme PNJ. Désigne un avatar représentant un personnage du monde virtuel dont le comportement est géré par l'application elle-même et non pas par un joueur.

Player killer: dans le jargon du jeu-vidéo, désigne un joueur dont la principale motivation est de ruiner le plaisir de jeu autres joueurs. Grand ennemi du *game-designer*, ce type de joueur exploitera toutes les failles du *game-play* .

Power levelling: dans le jargon du jeu-vidéo, technique qui consiste à utiliser un avatar d'un joueur pour faire obtenir à un avatar de moindre

compétence, de manière disproportionnée, des avantages qu'il n'aurait pas acquis en jouant de manière régulière comme l'ont imaginé les concepteurs du jeu.

Rollback: terme emprunté au domaine des bases de données. Dans le cadre d'un jeu massivement multi-joueurs, il désigne le retour en arrière à un état précédent et cohérent de l'application, en cas de corruption non correctible de l'état actuel. Le *rollback* est en général mal vécu par les joueurs qui peuvent y perdre le tribut de nombreuses heures de jeu.